EVALUATING INFORMATION LEAKAGE BY QUANTITATIVE AND INTERPRETABLE MEASUREMENTS

Ziqiao Zhou

A dissertation submitted to the faculty at the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2020

Approved by:

Michael Reiter

Jack Snoeyink

Montek Singh

David Evans

Ilya Mironov

ABSTRACT

Ziqiao Zhou: Evaluating Information Leakage by Quantitative and Interpretable Measurements
(Under the direction of Michael Reiter)

Noninterference, a strong security property for a computation process, informally says that the process output is insensitive to the value of its secret inputs – the secret inputs do not "interfere" with those outputs. This is too strong, however; a degree of interference is necessary in almost all real systems. In this dissertation, we propose a *measure* of noninterference that is more practical. Based on a model of computations with three types of input (secret, attacker-controlled, and others) and an attacker-observable output, we define a noninterference measure that can assess and explain information leaks in actual codebases.

We start with assessing a new defense against cache-based side-channel attacks in a cloud environment, using an experiment-based quantitative measure of leakage against existing attacks. It is not enough to measure leakage through empirical analysis, however, as it fails to identify new interference introduced by a weak defense design. We propose a symbolic execution framework to formally measure interference in simple software procedures, encompassing any interference from a set of secret inputs to observable outputs. Leveraging approximate model counting techniques, we make this framework scalable with parallelization. Unfortunately, this technique does not scale to support analysis of hardware processor designs, in part due to its reliance on symbolic execution to create a logical postcondition of the computation. We thus modified the framework to sidestep symbolic execution when analyzing processor designs. To further tame the complexity due to various sources of interference, we extend our framework to remove, or declassify, certain interference from consideration, so that the framework instead highlights other forms of interference, and to provide human-interpretable rules that explain the conditions under which interference occurs. We demonstrate the practicality of the work through case studies of both software-based leakage and vulnerabilities in the RISC-V BOOM core with different configurations.

To my family

ACKNOWLEDGEMENTS

Foremost, I would like to thank my committee members: Dr. Jack Snoeyink, Dr. Montek Singh, Dr. David Evans, Dr. Ilya Mironov for agreeing to trudge through my dissertation. I have my deepest appreciation to my advisor, Dr. Michael Reiter, for providing me enough support in my research. His endless enthusiasm and rigorous attitude always help me make our work better than expected. Our weekly meetings would be the most valuable moments. Being Mike's student is undoubtedly the correct choice I made.

I also want to acknowledge my collaborators and mentors. Although I am the only author of this dissertation, I get rid of "I" in chapters to credit my advisor and my collaborators' contributions. I thank Dr. Yinqian Zhang for sharing his research experience to help me get started and providing useful feedback about the system design, when he was a postdoctoral researcher in the same group and even after joining OSU. I thank to Dr. Zhiyun Qian for his active collaboration in static analysis and his insightful feedback in my case studies. Many thanks to our Intel contacts, Dr. Matthew Fernandez for his encourage in my research and his voluntary help in my job searching. Special thanks to my mentor Dr. Junghwan Rhee for his patient supervision in NEC Lab. The intern experience with him provides me enough chance to learn from machine learning experts, which highly broadens my research horizon. I appreciate my mentor, Dr. Michael Vrable, at Google, for providing me career advice and support. With his help, I learned the value of industry problems, which are easy for small systems but become tricky in large systems.

I thank my labmates and my friends at UNC-Chapel Hill. With their help, I do not feel that I am alone. In particular, thanks to Andrew Chi, Robert Cochran, Adam Humphries, Victor Heorhiadi, Jung Jiang, Sheng Liu, Marie Nesfield, Ke Coby Wang, and Qiuyu Xiao. I always learned a lot from their works from different perspectives.

A very special gratitude goes out to National Science Foundation and Intel for funding the work.

Completing my Ph.D. while leaving my hometown is not easy for me. The key making me go

so far with comfort is that I have my family, particularly my father, my wonderful stepmother, and my sister support and encourage me. I owe a lot to my grandmother, who raised me during my childhood. I thank Sheng Liu, who knows and supports me much no matter what happens. I could never have done this without them. I love you all.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

#∃SAT Projected Model Counting Problem. 13, 75, 76

CRIME Compression Ratio Info-leak Made Easy. xii, 35, 55–57, 63

LLC Last-level Caches. 1, 7, 14, 20, 22, 27, 30, 33, 100

SAT Satisfiability Problem. xiii, 13, 69, 71, 74–76, 78

#SAT Model Counting Problem. 12

CNF Conjunctive-normal-form. 12, 13

COA Copy-On-Access. 18, 19, 26

KSM Kernel Same-page Merging. 15, 17, 20

LFSR Linear-feedback Shift Register. 81

OS Operating System. 8, 15, 16

PaaS Platform-as-a-service. 15

PTE Page Table Entry. 17–20, 24–26

PTW Page Table Walker. 81

QIF Quantitative Information Flow. 1, 10–12, 40, 52, 101

VM Virtual Machine. 14

CHAPTER 1: INTRODUCTION

Unintended information leaks in resource-shared environments are a persistent problem in computer systems. The cause of those leaks is improper information flows that transfer information from secret variables to public channels. In part, such information leakage arises from developers' inability to test for them, since information leaks are typically not be evident in functional testing.

Static program analysis is one approach to discover information leaks before they occur. However, tools for static analysis suffer from a variety of shortcomings. First, most existing information-flow tools (e.g., taint analysis and model checking) do not measure leakage, but merely detect it. Second, due to the complexity of control-dependent assignments (i.e., implicit flows) and the lack of domain knowledge within the detector, detection tools raise false alarms.

Measuring is important since perfect security is usually not possible in the real world. For example, to defend against cache-based side channels, many mitigations [101, 93, 90] proposed in recent years only increase the difficulty of exploiting cache channels but do not fully close them. How to prove the security of those defenses is challenging, as some defenses may eliminate known information flows but open new ones. In Chapter 3, we design a software method called CacheBar to defend against cache-based side channels in last-level caches (`LLC`). The defense is proved to be effective using both attack-specific empirical tests and model checking. However, empirical tests do not capture some information flows not triggered by the tests performed, which are detected instead by automated model checking. That is, empirical tests provide lower coverage of leakage sources in the defense system than static methods.

Quantitative information flow (QIF) [29, 46] is proposed to measure the amount of leakage from a computation. Due to the computation complexity, existing QIF implementations (e.g., [96, 16, 77, 48]) tend to sacrifice measurement fidelity by using empirical data or abstract models, instead of static analysis on actual code bases. Furthermore, the measures used in earlier QIF works do not capture the leakage pattern. For example, entropy loss, a popular metric used in many QIF theoretical studies, only expresses the "expected" number of bits leaked, which can either leak

that number of bits in all executions or leak many more bits in some cases but few bits in others. Chapter 4 proposes a more refined measure of leakage to help developers measure interference patterns conditioned on sets of secret values. Chapter 5 extends the framework to accommodate computations on specific hardware processor designs, while providing ways to decompose causes of leakage and explain the causes using an interpretable model.

This dissertation is structured as follows. It first introduces some terminology to depict an end-to-end security model called noninterference in Sec. 1.1. The model guides us in defining and measuring leakage, starting with an empirical evaluation for cache-based side-channel attacks in Chapter 3 and then maturing to a static measurement framework in Chapter 4. In addition, Sec. 1.2 and Sec. 1.3 present the challenges in real-world noninterference evaluations, which are addressed by a modified framework that accommodates hardware designs in Chapter 5. We also present the toolchains implementing the proposed methodologies and use them to evaluate software and hardware computations. Those results together demonstrate the dissertation statement:

Information leakage from computation can be quantitatively measured and expressed in a human-interpretable model. Using this model, one can evaluate existing codebases and modifications thereof to restrict information flows.

1.1 Security Model

Noninterference [44] is a commonly used end-to-end model for information-flow security. In the noninterference model, a secured computation is seen as a machine having inputs and outputs where its secret does not determine what lower-level users (or attackers) can see or gain access to. In other words, the secret inputs do not "interfere" with those outputs. Motivated by the security model, this dissertation targets the information leaked about secret variables input to a *procedure proc*. More specifically, we divide the formal input parameters to procedure *proc* into three disjoint sets, namely $Vars_{\mathsf{C}}$, $Vars_{\mathsf{S}}$, and $Vars_{\mathsf{I}}$, having the following properties.

- $cvar \in Vars_{\mathsf{C}}$ takes on a value $\mathsf{C}(cvar)$ controlled by the attacker.
- $svar \in Vars_{\mathsf{S}}$ takes on a value $\mathsf{S}(svar)$ that is a secret for which an analyst is specifically concerned with detecting leakage via the outputs $Vars_{\mathsf{O}}$. Unless otherwise stated we will assume there is one secret input variable 'secret', and so $Vars_{\mathsf{S}} = \{\text{'secret'}\}$.
- $ivar \in Vars_{\mathsf{I}}$ takes on a value $\mathsf{I}(ivar)$ that is not controlled by the attacker. The use of *ivar* would be emphasized in Sec. 4.1.2.

In addition, there is a set $Vars_{\mathsf{O}}$ such that each $ovar \in Vars_{\mathsf{O}}$ takes on a value $\mathsf{O}(ovar)$ that is observable by the attacker. So we consider *proc* to be of the form

$$\mathsf{O} \leftarrow proc(\mathsf{C}, \mathsf{I}, \mathsf{S})$$

This dissertation assumes that *proc* is deterministic; a nondeterministic computation *proc* can be rendered deterministic by adding any randomness as inputs, say I('coins'). A given *proc* can then be characterized by a logical postcondition $\Pi_{proc}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{O})$ that constrains how the values in O relate to those in C, I, and S in any execution.

A poorly designed procedure (and thus postcondition) may expose information about the secret value in S through observable in O when the attacker chooses the input in C, which is the information leakage that our framework will focus on. That is, for one execution with any S and I, another execution with any S' must have a chance (with some C') to output equivalent O, when the attacker chooses any equivalent controlled input C. A formal definition of noninterference for a procedure *proc* could be expressed as:

$$\forall \mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{S}' : \exists \mathsf{I}', \mathsf{O} : \Pi_{proc}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{O}) \wedge \Pi_{proc}(\mathsf{C}, \mathsf{I}', \mathsf{S}', \mathsf{O})$$

However, noninterference is overly strict, excluding any computation outputting some observable values not feasible for some secret value. This dissertation instead explores the quantitative measure of noninterference. One possible direction is to measure interference based on empirical data, which is used in Chapter 3 to evaluate cache-based side channels in a complicated defense. Another direction is to analyze the victim's program through static analysis, which is discussed in Chapter 4 to cover more interference cases than empirical evaluations.

1.2 Information Declassification

In the real world, a computation may have inevitable leaks. For example, a password checker revealing whether the password is matched with attacker's input has an insecured flow from the confidential password to the output. Such insecured flow is due to the designed functionality and is an intended leakage.

Information declassification is an approach to exempt such allowed information flows [84], In-

formation flow research uses declassification to avoid reporting the leakage due to these allowed releases of information. However, those works only detect but do not measure additional leakage. Without exempting the declassified information from the leakage, a measure may incorrectly estimate the illegal leakage of a target program.

Chapter 5 proposes a methodology to use *declassification* in leakage measurement. The methodology enables analysts to *declassify* certain information, thereby focusing the measurement on any *other* leakage that might be occurring, i.e., leakage that cannot be inferred from the declassified information. For systems as complex as modern processors, this ability is essential to permit analysts to decompose and analyze leakage in a piecemeal fashion.

Slightly different from existing declassification research [83, 21], where declassification only reduces the leakage, our work better defines the actual unintended leakage with an awareness that declassification of certain information can either increase or decrease the unintended leakage. On the one hand, if some interference is due to declassification of a part of the secret, our measurement should not reflect that declassified information. For instance, suppose the procedure $proc(\mathsf{C},\mathsf{I},\mathsf{S})$ returns the lowest four bits of the secret:

$$proc(\mathsf{C},\mathsf{I},\mathsf{S}): \quad \mathsf{O}(\text{`result'}) \leftarrow \mathsf{S}(\text{`secret'}) \bmod 16$$

and the declassified information is whether the secret value $\mathsf{S}(\text{`secret'})$ is even or odd (i.e., $\mathsf{S}(\text{`secret'}) \bmod 2$). The interference measure should be decreased by this declassification, as the interference caused by the lowest bit has been allowed. On the other hand, if the declassification increases the risk to the secret when an attacker utilizes both the observable output from the computation and the allowed declassified information, then this increase should be reflected in the measure. For example, if the computation $proc(\mathsf{C},\mathsf{I},\mathsf{S})$ uses a random variable I ('coins'):

$$proc(\mathsf{C},\mathsf{I},\mathsf{S}): \quad \mathsf{O}(\text{`result'}) \leftarrow \mathsf{S}(\text{`secret'}) + \mathsf{I}(\text{`coins'})$$

where the declassified information is $\mathsf{I}(\text{`coins'})$, then it leaks more than than *proc* without declassification, since an attacker can use $\mathsf{O}(\text{`result'})$ and the declassified $\mathsf{I}(\text{`coins'})$ to reveal more about $\mathsf{S}(\text{`secret'})$. We demonstrate those effects using both dummy and actual computations in Chapter 5.

```
proc (C, I, S)
    if (C('test') mod 2 = 1)
        if( S('secret') mod 2 = 1)
                O('result') ← 1
        else
                O('result') ← 0
    else
        O('result') ← 1
```

(a) Procedure (implicit flow)

```
proc (C, I, S)
    O('result') ← C('test') & S('secret') & 1
```

(b) Procedure (explicit flow)

```
proc (C, I, S)
    O('result') ← C('test') & S('secret') & 2
```

(c) Procedure (different explicit flow)

Figure 1.1: Motivating examples

1.3 Interpreting Leakage

A computation includes both software and hardware implementations. The sheer complexity of hardware designs (e.g., CPU processor) means that once leakage is measured, the exact conditions that cause this leakage might not immediately be evident. Our work seeks a measure with interpretability to help developers understand sources of leakage and how to rectify them.

Chapter 4 provides a way to quantitatively measure the leakage for a procedure, which depicts how much about the secret is leaked due to the interference and how often an interference could happen. However, it does not explain *why* the procedure leaks information.

Consider the two procedures shown in Fig. 1.1(a) and Fig. 1.1(b). The procedure in Fig. 1.1(a) returns 1 if $\mathsf{C}(\text{`test'}) \bmod 2 = 0$ and $\mathsf{S}(\text{`secret'}) \bmod 2$ otherwise, and the second procedure in Fig. 1.1(b) returns the least significant bit of $\mathsf{S}(\text{`secret'}) \,\&\, \mathsf{C}(\text{`test'})$. As such, the two procedures leak the same information about the secret (i.e., both leak the least significant bit of the secret when $\mathsf{C}(\text{`test'})$ is odd and nothing otherwise), using different coding styles. Indeed, both procedures have the same quantitative leakage if we just *quantify* their interference. However, the quantitative measure does not express the concrete pairs of inputs causing the interference, thus losing some information about how the attacker's controlled and observable variables work together to reveal the secret. For example, the procedure in Fig. 1.1(c), which reveals the second bit of the secret when the second bit of $\mathsf{C}(\text{`test'})$ is 1, leaks the same amount of information about a different portion of the secret under a different attack condition. Merely relying on the quantitative measure proposed in Chapter 4, we cannot explain to the developer that Fig. 1.1(c) causes a different interference from Fig. 1.1(a) and Fig. 1.1(b).

Chapter 5 incorporates a method of *interpreting* the leakage, i.e., providing simple rules that

indicate circumstances in which leakage will (or will not) occur. Each such rule is additionally accompanied by a precision and recall, so that analysts can prioritize the rules they address.

1.4 Implementation Considerations

There are several different ways to try to quantitatively measure interference. Depending on the targeted computations, we may choose different implementations to quantify the leakage.

One possible method to evaluate the security of a complicated system is to replay existing attacks and measure the difficulty of revealing the secret using the attacks. In Chapter 3, we illustrate this approach by collect the attacker's ability to exploit cache-based side-channel attacks in a machine with or without a defense we propose. We train a classifier to tell how many cache lines are used by the victim during a computation and calculate a confusion matrix to assess whether an attacker can distinguish a secret value in the victim program from others, by using this classifier. Since the measurement is based on data collected from specific attacks, it does not guarantee the leakage's completeness under all conditions.

To cover more sources of interference, Sec. 4.2 proposes a static quantification method using symbolic execution to extract the logic postcondition Π_{proc} of *proc*. With Π_{proc}, that chapter explores the assessment of leakage vulnerabilities by randomly sampling a space of secret values and then limiting our search for pairs of attacker-controlled inputs and attacker-observable outputs to only those that are consistent with some secret in that space. Finding two spaces of secret values for which these counts suggest pairs consistent with one but not both then reveals interference.

To evaluate joint hardware-software vulnerabilities, static analysis (e.g., through symbolic execution) is unscalable due to the complexity of the hardware. Sec. 5.3 describes a implementation called DINoMe to statically evaluate hardware-software vulnerabilities. DINoMe targets software snippets for up to hundreds of cycles in open-sourced CPU processors (e.g., RISC-V BOOM). Specifically, DINoMe considers a *proc* composed with partially symbolic processor and assembly (i.e., software), which could complete execution within hundreds of CPU cycles. Instead of symbolically executing this *proc* for multiple cycles (e.g., [97]), which should be expensive, DINoMe generates a transition logic from the current state to the next-cycle state and then efficiently stitches together a multi-cycle postcondition. Although the dissertation starts with the CacheBar work and ends with a static methodology for measuring interference, the limitations of DINoMe (discussed in Sec. 5.5) make it hard to adapt to CacheBar.

CHAPTER 2: BACKGROUND AND RELATED WORK

2.1 Side Channels in CPU Caches

Cache-based side channels are important attack vectors that have been researched for many years (e.g., [99, 95, 68]). The most common cache-based side-channel attacks are Prime+Probe, Flush+Reload, and their variants.

2.1.1 Flush+Reload

The Flush+Reload (e.g., [99, 95]) is a highly effective cache-based side channel attacks that was used, e.g., by Zhang et al. [99], to mount fine-grained side channels when memory sharing enabled. It leverages physical memory pages shared between an attacker and victim security domains (e.g., due to shared libraries), as well as the ability to evict those pages from cache, using a capability such as provided by the `clflush` instruction on the x86 architecture. In the flushing stage, the attacker Flushes a chunk of the shared memory out of the cache. After a short time interval (the "Flush+Reload interval") during which the victim executes, the attacker measures the time to Reload the same chunk. In this case, the secret data whose value is reflected in the use of shared memory is `S`, while the cache hit or miss which is reflected by the timing of Reload is mapped to `O`.

2.1.2 Prime+Probe

Another common method to launch side-channel attacks via caches is using Prime+Probe attacks, introduced by Osvik et al. [73]. These attacks have recently been adapted to use `LLC`s to great effect, e.g., [68, 50]. Unlike a Flush+Reload attack, Prime+Probe attacks do not require the attacker and victim security domains to share pages. Here, the attacker utilizes the architectural property of an associative cache that different memory blocks may be stored in the same cache set. In the Prime stage, the attacker Primes the cache by loading memory blocks into a target cache set. Subsequently, the attacker Probes the cache by measuring the time to access the previously loaded memory blocks. The time to do so tells the attacker how many cache lines in that cache set were evicted by the victim in the interim. Generally, secret data (e.g., the private key

in decryption) whose value decides the use of memory blocks (e.g., indices in some lookup tables, or the use of key-dependent instructions) mapping to known cache sets is S, while the cache hit or miss which is reflected by the timing of PROBE is mapped to O.

2.1.3 Speculative Execution CPU Risk

The cause of cache-based side channel attacks is the poorly controlled information flow from sensitive data/instructions to shared cache resources. The recent Spectre [57] and Meltdown [65] attacks further demonstrate the joint risk of speculative execution and cache side channels , in this case allowing an attacker to read arbitrary memory locations in a victim process. To provide more concurrency, a CPU predicts the outcome of a conditional branch and executes instructions based on that prediction to reduce delays incurred by those instructions if its prediction was correct. However, even if the prediction is incorrect, then some changes to the cache caused by speculative execution will persist even after the mispredicted computations have been discarded. Those changes propagate unintended information to exploitable cache-based side channels, allowing the attacker to steal them.

2.1.4 Mitigations and their proof of security

Numerous proposals have sought to mitigate cache-based side channels in application, system, and hardware levels.

The straightforward method to remove a cache-based side channel for a specific application is to modify the application's software code to better protect secrets from side-channel attacks. These solutions range from tools to limit branching on sensitive data (e.g., [27, 28]) to application-specific side-channel-free implementations (e.g., [58]). However, the the overheads of these techniques tend to grow with the scope of programs to which they apply and can be very substantial (e.g., [81]).

Hardware-based mitigations redesign the the cache with a stronger isolation between different security domains. One direction of the new design is to manage ownership of a cache line so that disallowing the interference from unauthorized users e.g., [92, 54, 67]). Other works (e.g., [93, 90]) tends to use domain-specific memory-to-cache mapping to increase the difficulty to decrypt the memory address. However, deploying secured cache design to all machines is not that possible in the foreseeable future.

System-level countermeasures tend to mitigate the cache side channels by modifying operating system (OS) or hypervisor. The modifications obfuscate the cache timing or isolate the memory

access across different security domains. For example, several works provide to each security domain a limited number of designated memory that are never evicted from the LLC (e.g., [55, 66]), thereby rendering their contents immune to PRIME+PROBE side channel attacks. To mitigate FLUSH+RELOAD, some works [74, 99, 11] have suggested disabling or selectively enabling memory sharing for countering various side-channel attacks exploiting shared memory, while stopping short of exploring a complete design for doing so.

Those works usually use theoretical explanations [55, 92] or empirical evaluations to prove their improved security. A commonly used empirical evaluation targets the timing channel strength in a specific cryptographic system (e.g., [27, 58, 54, 67, 66]). Specifically, they rerun concrete attacks targeting a cryptographic system and then use timing statistics (e.g., the timing difference) to reflect the timing channel strength. Such evaluations do not reflect the attacker's actual power using a classifier, which takes timing as input and predicts the secret. Another direction to evaluate the cache-based side channels depicts the side-channel leakage through the performance of the classifier, e.g., the expected number of correct bits (e.g., [28]) or the confusion matrix [81]). In Chapter 3, we provide a copy-on-access design as an efficient memory isolation selectively enabling memory sharing for addressing FLUSH+RELOAD attacks, and extend this idea with cacheability management for PRIME+PROBE defense, as well. Then we prove the improved security using confusion matrices from empirical evaluations and further check the security using a formal model.

2.2 Generalization of Noninterference Property

Noninterference property provides a strong security guarantee of zero information leakage. *Security type system*s enforce the noninterference by tracking information flow within programs and checking security rules used to assign variables with different security labels. However, strict noninterference makes those systems hard to use in practice due to inevitable flows from high-security to low-security labels. To make the noninterference model more tractable, *declassification* [84] provides a weaker form of information flow policies that defines *what* information could be released, *where* within the code it is released (i.e., locations), *who* releases it (i.e., users), and *when* it is released (i.e., time). To support declassification, many works (e.g., [83, 43, 21, 9, 38]) changes security type systems to include a means of downgrading to permit high-security data to propagate to low-security variables.

2.2.1 Delimited release

Sabelfeld et al. [83] introduces a method to define declassification use called *delimited release*. It uses a specialized function `declassify` to mark a collection of expressions about *what* information to release. Then a program satisfies the delimited release if it has the following property: for any two executions which only differs in the value of secret S and S', observable value is always the same when the value of expressions defined by `declassify` function is the same.

In chapter Chapter 5, a modified quantitative measure accepts the declassified information which defines *what* a secret information could be released through a developer-defined logic formula.

2.2.2 Abstract noninterference

Giacobazzi et al. [42, 43] introduce the notion of abstract noninterference to weaken the noninterference by parameterizing standard noninterference relatively to what an attacker can observe. Specifically, the abstract noninterference considers properties instead of values as observable objects. Since the attacker may not be able to access all public values in a program directly, the abstract noninterference intuitively defines a *weaker observer* (than an observer directly using values from the victim's procedure) through abstract interpretation. To use declassification, it again uses an input property representing which inputs they need to check noninterference.

The case studies for cache-based side channels in Chapter 5 compose attacker's observable and controllable variables instead of directly using existing variables in the cache module, as an attacker only loads or flushes memory blocks and observes cache hit or miss but does not directly modify or observe any registers in cache modules.

2.3 Quantitative Information Flows

Another direction to generalize noninterference property is to measure instead of only detecting the violation. First introduced by Denning [29] and Gray [46] in the 1980s, QIF measures the amount of information leaked about a secret by observing a program execution.

2.3.1 Measuring entropy uncertainty

The earliest model of QIF ([29, 46, 22, 23, 24]) uses the Shannon mutual information to measure uncertainty about the secret. Smith [86] claims that Shannon entropy fails to accurately capture the vulnerability of a program, as it does not measure the probability of correct guessing. Furthermore, Clarkson et al. [25] argues that the uncertainty-based measurement is inadequate as it only measures

the probability of an attacker being absolutely correct or absolutely wrong.

Our work improves on prior work in QIF along one or more of the following dimensions. First, computing the measures in these works often involves computing outputs induced by sampled secret values (e.g., [24]), which sometimes leverages application-specific restrictions to be tractable (e.g., [96]). Our framework proposed in Chapter 4, in contrast, does not require such application-specific restrictions. Second, exploiting leakage vulnerabilities often requires attackers not only to observe outputs but also to inject inputs, and many applications incorporate other inputs, as well. These QIF calculations are not possible without knowing the distributions from which these values are drawn (e.g., [71]), and so some works (e.g., [59, 78]) heuristically assign specific values to these unknown inputs, potentially hiding the leakage from other assignments. Our analysis computes conditionals in a different "direction," i.e., counting possible combinations of attacker-controlled inputs and attacker-observable outputs conditioned on sets of secret values and while leaving other inputs constrained. In doing so, our technique accommodates attacker-controlled inputs but does not presume knowledge of the attacker's strategy or the distributions of these or other inputs. Third, some QIF frameworks work only for deterministic procedures (e.g., [76, 60]), whereas ours accommodates nondeterministic ones, as well.

2.3.2 Differential privacy

Differential privacy [33] is a criterion for privacy protection that many algorithms have been devised to satisfy. As originally expressed, differential privacy requires that any output observed from a computation on a database is insensitive to the existence of any single element (row) in that database; i.e., the probability of observing a computation output is nearly the same even if any single row is added or removed. Viewing the database as our secret value, this definition therefore requires that computations on *nearby* secrets result in the same attacker-observable outputs, with high probability. In contrast to differential privacy, our work does not leverage a distance measure over secrets; i.e., there is no notion of "nearby" secrets in our definition.

Moreover, the focus of our work is somewhat different in providing a way to measure and explain leakage for arbitrary software or hardware designs, versus in providing a prescriptive measure to limit that leakage. Notably, since differential privacy requires a statistical guarantee of indistinguishability for *any* two nearby databases, it mandates a requirement that typically can be met only through the addition of noise artificially to the computation output. As such, this definition

has driven considerable research on algorithms for adding noise to observable outputs to enforce this condition.

2.4 Model counting

Solution counting, the problem of computing the number of solutions for a given constraint, is necessary for static analysis of QIF, which does not rely on empirical data. For a propositional formula, the counting problem is called model counting problem (`#SAT`) [45], where a model is a feasible solution for F. Thus, it is a #NP problem. Practical model counting techniques can be categorized to *exact counting* and *approximate counting*.

Exact model counting tends to use DPLL-style exhaustive search. Specifically, it uses a backtracking algorithm to repeatedly search a feasible model, block it, and find a new model. Such exhaustive technique is not scalable when the number of models is large. *Approximate model counting* uses a sampling method to estimate the number of feasible models in the formula without enumerating all models. The recent hash-based model counting proposed by Chakraborty et al. [15] improves the scalability and provides a proven lower and upper bounds with a confidence guarantee in a statistical sense.

2.4.1 Hash-based approximate model counting

The hash-based approximate model counting technique due to Chakraborty et al. [15] leverages a family of 3-wise independent hash functions to estimate the number $\#F$ of satisfying assignments of a conjunctive-normal-form (CNF) proposition F of v variables and runs in fully polynomial time with respect to a SAT oracle. At a high level, this algorithm iteratively selects a random hash function $H^b : \{0,1\}^v \rightarrow \{0,1\}^b$ from a family (where b changes per iteration) and a random $p \in \{0,1\}^b$, and computes the satisfying assignments for F for which the hash of the assignment (a string in $\{0,1\}^v$) is p. (Intuitively, this number should be about a $\#F/2^b$.) Through judicious management of this iterative process, the algorithm arrives at an estimate $\tilde{\#}F$ for $\#F$ that satisfies

$$\mathbb{P}\left((1+\epsilon)^{-1} \cdot \#F \leq \tilde{\#}F \leq (1+\epsilon) \cdot \#F\right) \geq \delta$$

where error ϵ, $0 < \epsilon \leq 1$, and confidence δ, $0 < \delta \leq 1$, are parameters and the probability is taken with respect to the random choices of the algorithm.

Previous QIF-related works leveraging model counting either support only convex constraints

(e.g., [7, 76]) and so therefore do not capture all constraints of realistic applications, or use exact counters (e.g., [77]) and so cannot scale to complex applications. In contrast, Chapter 4 leverage principled sampling-based methods for counting purposes, which we show can be used to expose leaks in real codebases. Chapter 4 also demonstrates a new approach for using model counting to estimate information leakage based on noninterference property, again deriving from a strategy of counting pairs of attacker-controlled inputs and attacker-observable outputs conditioned on secret value sets of different sizes, in contrast to these prior works.

2.4.2 Projected model counting

Projected model counting problem (#∃SAT) counts feasible assignments of selected variables in a propositional formula. For a realistic application, the automatically generated CNF formula would introduce auxiliary variables in order to support the encoding of different operations. Thus, our counting task is projected model counting. The algorithm used in model counting can be applied to project model counting through a minor modification. In Chapter 4, instead of applying a hash constraint to all variables in the F, the implementation adds a hash function over the counting targets C,I, and O.

CHAPTER 3: A DEFENSE AGAINST CACHE-BASED SIDE CHANNELS WITH EMPIRICAL SECURITY[1]

This chapter focuses on information leakage through the cache-based side channels in LLCs. We expect to use this dedicated example to emphasize the importance of measuring with static analysis, by comparing the empirical measure and static verification based on a formal model.

Recall that FLUSH+RELOAD, PRIME+PROBE, and their variants are the most commonly used attack vectors in cache-based side channels. We propose a software-only defense against these LLC-based side-channel attacks, based on two seemingly straightforward principles. First, to defeat FLUSH+RELOAD attacks, we propose a copy-on-access mechanism to manage physical pages shared across mutually distrusting *security domains* (i.e., processes, containers[2], or virtual machines (VMs)). Specifically, temporally proximate accesses to the same physical page by multiple security domains results in the page being copied so that each domain has its own copy. In this way, a victim's access to its copy will be invisible to an attacker's RELOAD in a FLUSH+RELOAD attack. When accesses are sufficiently spaced in time, the copies can be deduplicated to return the overall memory footprint to its original size. Second, to defeat PRIME+PROBE attacks, we design a mechanism to manage the cacheability of memory pages so as to limit the number of lines per cache set that an attacker may PROBE. In doing so, we limit the attacker's visibility into the victim's demand for memory that maps to that cache set.

Of course, the essential part of defense work is to prove their effectiveness through a reasonable security evaluation. To do so, we first detail design and implementation in a memory management subsystem called CACHEBAR (short for "Cache Barrier") for the Linux kernel. CACHEBAR supports these defenses for security domains represented as Linux containers. That is, copy-on-access to defend against FLUSH+RELOAD attacks makes page copies as needed to isolate temporally

[1]This chapter is excerpted from previously published work [101] coauthored with Michael K. Reiter and Yinqian Zhang.
[2]https://linuxcontainers.org/

proximate accesses to the same page from different containers. Moreover, memory cacheability is managed so that the processes in each container are collectively limited in the number of lines per cache set they can PROBE. CACHEBAR would thus be well-suited for use in platform-as-a-service (PaaS) clouds that isolate cloud customers in distinct containers. With a concrete implementation, we design a quantitative empirical evaluation to measure the attacker's ability to distinguish a victim's behavior in a PaaS cloud environment. Our empirical results show that CACHEBAR effectively restricts the leakage in cache-based side-channel attacks. Besides, we build a formal model for copy-on-access and use model checking to check potential interference. The checking results reveal the incompleteness of empirical evaluation, which covers more vulnerable information flows than rerunning concrete attacks.

3.1 Copy-On-Access

3.1.1 Design

Modern operating systems, in particular Linux OS, often adopt on-demand paging and copy-on-write mechanisms to reduce the memory footprints of userspace applications. In particular, copy-on-write enables multiple processes to share the same set of physical memory pages as long as none of them modify the content. If a process writes to a shared memory page, the write will trigger a page fault and a subsequent new page allocation so that a private copy of page will be provided to this process. In addition, memory merging techniques like kernel same-page merging (KSM) [5] are also used in Linux OS to deduplicate identical memory pages. Memory sharing, however, is one of the key factors that enable FLUSH+RELOAD side-channel attacks. Disabling memory page sharing entirely will eliminate FLUSH+RELOAD side channels but at the cost of much larger memory footprints and thus inefficient use of physical memory.

CACHEBAR adopts a design that we call copy-on-access, which dynamically controls the sharing of physical memory pages between security domains. We designate each physical page as being in exactly one of the following states: UNMAPPED, EXCLUSIVE, SHARED, and ACCESSED. An UNMAPPED page is a physical page that is not currently in use. An EXCLUSIVE page is a physical page that is currently used by exactly one security domain, but may be shared by multiple processes in that domain. A SHARED page is a physical page that is shared by multiple security domains, i.e., mapped by at least one process of each of the sharing domains, but no process in any domain has

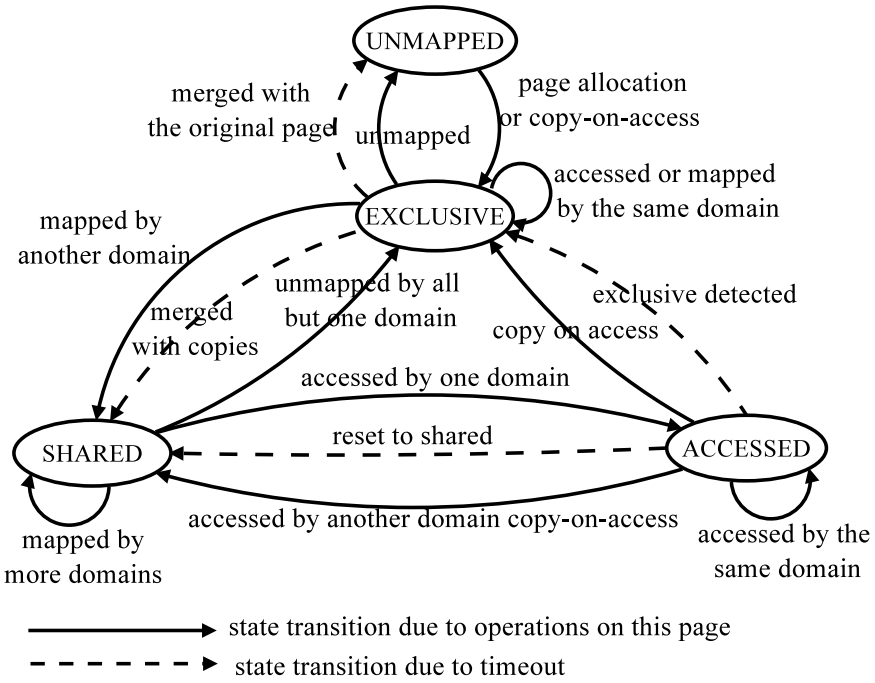


Figure 3.1: Copy-On-Access State Transition

accessed this physical page recently. In contrast, an ACCESSED page is a previously SHARED page that was recently accessed by a security domain. The state transitions are shown in Fig. 3.1.

An UNMAPPED page can transition to the EXCLUSIVE state either due to normal page mapping, or due to copy-on-access when a page is copied into it. Unmapping a physical page for any reason (e.g., process termination, page swapping) will move an EXCLUSIVE page back to the UNMAPPED state. However, mapping the current EXCLUSIVE page by another security domain will transit it into the SHARED state. If all but one domain unmaps this page, it will transition back from the SHARED state to the EXCLUSIVE state, or ACCESSED state to the EXCLUSIVE state. A page in the SHARED state may be shared by more domains and remain in the same state; when any one of the domains accesses the page, it will transition to the ACCESSED state. An ACCESSED page can stay that way as long only the same security domain accesses it. If this page is accessed by another domain, a new physical page will be allocated to make a copy of this one, and the current page will transition to either EXCLUSIVE or SHARED state, depending on the remaining number of domains mapping this page. The new page will be assigned state EXCLUSIVE. An ACCESSED page will be reset to the SHARED state if it is not accessed for ΔT_{acc} seconds. This timeout mechanism ensures that only recently used pages will remain in the ACCESSED state, limiting chances for unnecessary

duplication. Page merging may also be triggered by deduplication services in a modern OS (e.g., KSM in Linux). This effect is reflected by a dashed line in Fig. 3.1 from state EXCLUSIVE to SHARED. A page at any of the *mapped* states (i.e., EXCLUSIVE, SHARED, ACCESSED) can transition to UNMAPPED state for the same reason when it is a copy of another page (not shown in the figure).

Merging duplicated pages requires some extra bookkeeping. When a page transitions from UNMAPPED to EXCLUSIVE due to copy-on-access, the original page is tracked by the new copy so that CACHEBAR knows with which page to merge it when deduplicating. If the original page is unmapped first, then one of its copies will be designated as the new "original" page, with which other copies will be merged in the future. The interaction between copy-on-access and existing copy-on-write mechanisms is also implicitly depicted in Fig. 3.1: Upon copy-on-write, the triggering process will first *unmap* the physical page, possibly inducing a state transition (from SHARED to EXCLUSIVE). The state of the newly mapped physical page is maintained separately.

3.1.2 Implementation

At the core of copy-on-access implementation is the state machine depicted in Fig. 3.1.

unmapped ⇔ exclusive ⇔ shared Conventional Linux kernels maintain the relationship between processes and the physical pages they use. However, CACHEBAR also needs to keep track of the relationship between containers and the physical pages that each container's processes use. Therefore, CACHEBAR incorporates a new data structure, `counter`, which is conceptually a table used for recording, for each physical page, the number of processes in each container that have page table entrys (page table entries) mapped to this page.

The `counter` data structure is updated and referenced in multiple places in the kernel. Specifically, in CACHEBAR we instrumented every update of `_mapcount`, a data field in the `page` structure for counting PTE mappings, so that every time the kernel tracks the PTE mappings of a physical page, `counter` is updated accordingly. The use of `counter` greatly simplifies maintaining and determining the state of a physical page: (1) Given a container, access to a single cell suffices to check whether a physical page is already mapped in the container. This operation is very commonly used to decide if a state transition is required when a page is mapped by a process. Without `counter`, such an operation requires performing reverse mappings to check the domain of each mapping. (2) Given a physical page, it takes N accesses to `counter`, where N is the total number of containers,

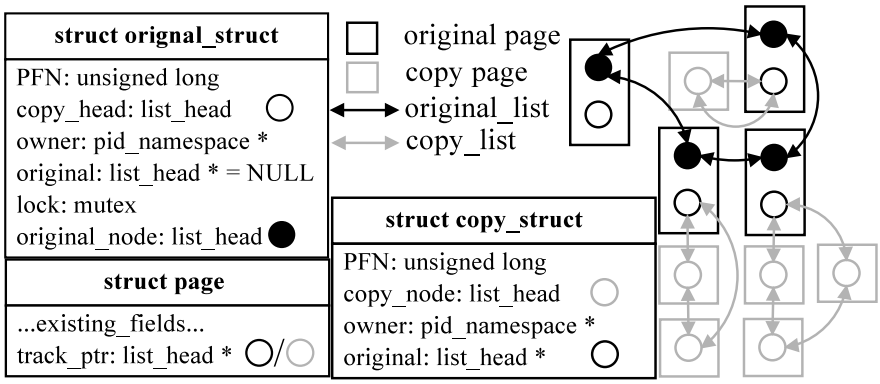


Figure 3.2: Structure of copy-on-access page lists.

to tell which containers have mapped to this page. This operation is commonly used to determine the state of a physical page.

shared ⇒ accessed To differentiate SHARED and ACCESSED states, one additional data field, `owner`, is added (see Fig. 3.2) to indicate the owner of the page (a pointer to a `PID_namespace` structure). When the page is in the SHARED state, its `owner` is `NULL`; otherwise it points to the container that last accessed it.

All PTEs pointing to a SHARED physical page will have a reserved Copy-On-Access (COA) bit set. Therefore, any access to these virtual pages will induce a page fault. When a page fault is triggered, CacheBar checks if the page is present in physical memory; if so, and if the physical page is in the SHARED state, the COA bit of the current PTE for this page will be cleared so that additional accesses to this physical page from the current process will be allowed without page faults. The physical page will also transition to the ACCESSED state.

accessed ⇒ exclusive/shared If the page is already in the ACCESSED state when a domain other than the `owner` accesses it, the page fault handler will allocate a new physical page, copy the content of the original page into the new page, and change the PTEs in the accessing container so that they point to the new page. Since multiple same-content copies in one domain burdens both performance and memory but contributes nothing for security, the fault handler will reuse a copy belonging to that domain if it exists. After copy-on-access, the original page can either be EXCLUSIVE or SHARED. All copy pages are anonymous-mapped, since only a single file-mapped page for the same file section is allowed.

A transition from the ACCESSED state to SHARED or EXCLUSIVE state can also be triggered

by a timeout mechanism. CacheBar implements a periodic timer (every $\Delta T_{\mathsf{acc}} = 1\text{s}$). Upon timer expiration, all physical pages in the ACCESSED state that were not accessed during this ΔT_{acc} interval will be reset to the SHARED state by clearing its `owner` field, so that pages that are infrequently accessed are less likely to trigger copy-on-access. If an ACCESSED page is found for which its `counter` shows the number of domains mapped to it is 1, then the daemon instead clears the COA bit of all PTEs for that page and marks the page EXCLUSIVE.

Instead of keeping a list of ACCESSED pages, CacheBar maintains a list of pages that are in either SHARED or ACCESSED state, denoted `original_list` (shown in Fig. 3.2). Each node in the list also maintains a list of copies of the page it represents, dubbed `copy_list`. These lists are attached onto the `struct page` through `track_ptr`. Whenever a copy is made from the page upon copy-on-access, it is inserted into the `copy_list` of the original page. Whenever a physical page transitions to the UNMAPPED state, it is removed from whichever of `original_list` or `copy_list` it is contained in. In the former case, CacheBar will designate a copy page of the original page as the new original page and adjust the lists accordingly.

For security reasons that will be explained in Sec. 3.3.1(a) , we further require flushing the entire memory page out of the cache after transitioning a page from the ACCESSED state to the SHARED state due to this timeout mechanism. This page-flushing procedure is implemented by issuing `clflush` on each of the memory blocks of any virtual page that maps to this physical page.

State transition upon `clflush` The `clflush` instruction is subject to the same permission checks as a memory load, will trigger the same page faults, and will similarly set the ACCESSED bit in the PTE of its argument [49]. As such, each Flush via `clflush` triggers the same transitions (e.g., from SHARED to ACCESSED, and from ACCESSED to an EXCLUSIVE copy) as a Reload in our implementation, meaning that this defense is equally effective against both Flush+Reload and Flush+Flush [47] attacks.

Page deduplication To mitigate the impact of copy-on-access on the size of memory, CacheBar implements a less frequent timer (every $\Delta T_{\mathsf{copy}} = 10 \times \Delta T_{\mathsf{acc}}$ seconds) to periodically merge the page copies with their original pages. Within the timer interrupt handler, `original_list` and each `copy_list` are traversed similarly to the "ACCESSED $\Rightarrow$ SHARED" transition description above, though the ACCESSED bit in the PTEs of only pages that are in the EXCLUSIVE state are checked.

If a copy page has not been accessed since the last such check (i.e., the ACCESSED bit is unset in all PTEs pointing to it), it will be merged with its original page (the head of the `copy_list`). The ACCESSED bit in the PTEs will be cleared afterwards.

When merging two pages, if the original page is anonymous-mapped, then the copy page can be merged by simply updating all PTEs pointing to the copy page to instead point to the original page, and then updating the original page's reverse mappings to include these PTEs. If the original page is file-mapped, then merging is more intricate, additionally involving the creation of a new virtual memory area (`vma` structure) that maps to the original page's file position and using this structure to replace the virtual memory area of the (anonymous) copy page in the relevant task structure.

For security reasons, merging of two pages requires flushing the original physical page from the `LLC`. We will elaborate on this point in Sec. 3.3.1(a) .

Interacting with KSM Page deduplication can also be triggered by existing memory deduplication mechanisms (e.g., KSM). To maintain the state of physical pages, CacheBar instruments every reference to `_mapcount` within KSM and updates `counter` accordingly. KSM is capable of merging more pages than our built-in page deduplication mechanisms. However, CacheBar still relies on the built-in page deduplication mechanisms for several reasons. First, KSM can merge only anonymous-mapped pages, while CacheBar needs to frequently merge an anonymous-mapped page (a copy) with a file-mapped page (the original). Second, KSM may not be enabled in certain settings, which will lead to ever growing `copy_list`s. Third, KSM must compare page contents byte-by-byte before merging two pages, whereas CacheBar deduplicates pages on the same `copy_list`, avoiding the expensive page content comparison.

3.2 Cacheability Management

A potentially effective countermeasure to Prime+Probe attacks is to remove the attacker's ability to Prime and Probe the whole cache set and to predict how a victim's demand for that set will be reflected in the number of evictions from that set.

3.2.1 Design

Suppose a w-way set associative `LLC`, so that each cache set has w lines. Let x be the number of cache lines in one set that the attacker observes having been evicted in a Prime+Probe interval

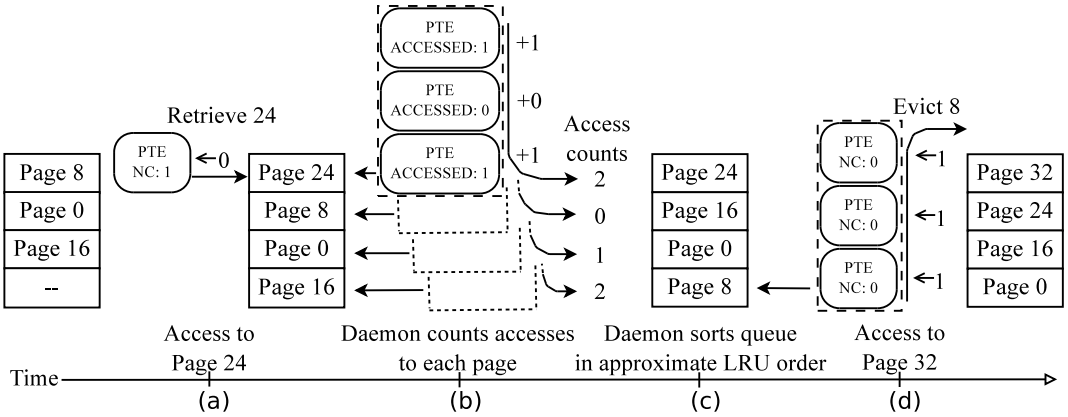


Figure 3.3: A cacheable queue for one page color in a domain: (a) access to page 24 brings it into the queue and clears NC bit ("$\leftarrow 0$") in the PTE triggering the fault; periodically, (b) a daemon counts the ACCESSED bits ("+0", "+1") per page and (c) reorders pages accordingly; to make room for a new page, (d) NC bits in PTE pointing to the least recently used page are set, and the page is removed from the queue.

(i.e., $x \in Vars_{\mathsf{O}}$). The PRIME+PROBE attack is effective today because x is typically a good indicator of the demand d that the victim security domain had for memory that mapped to that cache set during the PRIME+PROBE interval (i.e., $d \in Vars_{\mathsf{S}}$). In particular, if the attacker PRIMEs and PROBEs all w lines, then it can often observe the victim's demand d exactly, unless $d > w$ (in which case the attacker learns at least $d \geq w$).

Here we propose to periodically and probabilistically reconfigure the budget k_i of lines per cache set that the security domain i can occupy. After such a reconfiguration, the attacker's view of the victim's demand d is clouded by the following three effects. First, if the attacker is allotted a budget $k_{\mathsf{a}} < w$, then the attacker will be unable to observe any evictions at all (i.e., $x = 0$) if $d < w - k_{\mathsf{a}}$.[3] Second, if the victim is given allotment k_{v}, then any two victim demands d, d' satisfying $d > d' \geq k_{\mathsf{v}}$ will be indistinguishable to the attacker. Third, the probabilistic assignment of k_{v} results in extra ambiguity for the attacker, since x evictions might reflect the demand d or the budget k_{v}, since $x \leq \min\{d, k_{\mathsf{v}}\}$ (if all x evictions are caused by the victim).

To enforce the budget k_i of lines that security domain i can use in a given cache set, CACHEBAR maintains for each cache set a queue per security domain that records which memory blocks are

[3]This statement assumes a LRU replacement policy and that the victim is the only security domain that runs in the PRIME+PROBE interval. If it was not the only security domain to run, then the ambiguity of the observable evictions will additionally cause difficulties for the attacker.

presently cacheable in this set by processes in this domain. Each element in the queue indicates a memory block that maps to this cache set; only blocks listed in the queue can be cached in that set. The queue is maintained with a least recently used (LRU) replacement algorithm. That is, whenever a new memory block is accessed, it will replace the memory block in the corresponding queue that is the least recently used.

3.2.2 Dynamic budget k_i of cache lines

Suppose there are (at most) m domains on a host that are owned by the attacker—which might be all domains on the host except the victim—and let w be the number of cache lines per `LLC` set. Below we consider domain 0 to be the "victim" domain being subjected to PRIME+PROBE attacks by the "attacker" domains $1, \ldots, m$. Of course, the attacker domains make use of all $\sum_{i=1}^{m} k_i$ cache lines available to them for conducting their PRIME+PROBE attacks.

Periodically, CACHEBAR draws a new value k_i for each security domain i. This drawing is memoryless and independent of the draws for other security domains. Let K_i denote the random variable distributed according to how k_i is determined. The random variables that we presume can be observed by the attacker domains include $K_1, \ldots, K_m$; let $K_{\mathsf{a}} = \min\{w, \sum_{i=1}^{m} K_i\}$ denote the number of cache lines allocated to the attacker domains. We also presume the attacker can accurately measure the number X of its cache lines that are evicted during the victim's execution.

Let $\mathbb{P}_d(E)$ denote the probability of event E in an execution period during which the victim's cache usage would populate d lines (of this color) if it were allowed to use all w lines, i.e., if $k_0 = w$. We (the defender) would like to distribute $K_0, \ldots, K_m$ so as to minimize the statistical distance between eviction distributions observable by the attacker for different victim demands d, d', i.e., to minimize

$$\sum_{0 \le d < d' \le w} \sum_{x} \left| \mathbb{P}_d(X = x) - \mathbb{P}_{d'}(X = x) \right| \tag{3.1}$$

We begin by deriving an expression for $\mathbb{P}_d(X = x)$. Below we make the conservative assumption that all evictions are caused by the victim's behavior; in reality, caches are far noisier. We first

consider the case $x = 0$, i.e., that the attacker domains observe no evictions.

$$\mathbb{P}_d\Big(X = 0 \;\Big|\; K_0 = k_0 \wedge\ K_{\mathsf{a}} = k_{\mathsf{a}}\Big) = \begin{cases} 1 \;\text{ if } w \geq k_{\mathsf{a}} + \min\{k_0, d\} \\ 0 \;\text{ otherwise} \end{cases}$$

"$\min\{k_0, d\}$" is used above because any victim demand for memory blocks that map to this cache set beyond k_0 will back-fill the cache lines invalidated when CacheBar flushes other blocks from the victim's cacheability queue, rather than evicting others. Since K_0 and K_{a} are independent,

$$\mathbb{P}_d\left(X = 0\right) = \sum_{k_0=0}^{d} \sum_{k_{\mathsf{a}}=0}^{w-k_0} \mathbb{P}\left(K_0 = k_0\right) \cdot \mathbb{P}\left(K_{\mathsf{a}} = k_{\mathsf{a}}\right) + \sum_{k_0=d+1}^{w} \sum_{k_{\mathsf{a}}=0}^{w-d} \mathbb{P}\left(K_0 = k_0\right) \cdot \mathbb{P}\left(K_{\mathsf{a}} = k_{\mathsf{a}}\right) \quad (3.2)$$

Note that we have dropped the "d" subscripts from the probabilities on the right, since K_0 and K_{a} are distributed independently of d. And, since $K_1, \ldots, K_m$ are independent,

$$\mathbb{P}\left(K_{\mathsf{a}} = k_{\mathsf{a}}\right) = \begin{cases} \displaystyle\sum_{k_1+\ldots+k_m=k_{\mathsf{a}}} \prod_{i=1}^{m} \mathbb{P}\left(K_i = k_i\right) & \text{if } k_{\mathsf{a}} < w \\ \displaystyle\sum_{k_1+\ldots+k_m\geq w} \prod_{i=1}^{m} \mathbb{P}\left(K_i = k_i\right) & \text{if } k_{\mathsf{a}} = w \end{cases} \quad (3.3)$$

Similarly, for $x \geq 1$,

$$\mathbb{P}_d\Big(X = x \;\Big|\; K_0 = k_0 \wedge K_{\mathsf{a}} = k_{\mathsf{a}}\Big) = \begin{cases} 1 & \text{if } x + w = k_{\mathsf{a}} + \min\{k_0, d\} \\ 0 & \text{otherwise} \end{cases}$$

and so for $x \geq 1$,

$$\mathbb{P}_d\left(X = x\right) = \sum_{k_0=0}^{d} \mathbb{P}\left(K_0 = k_0\right) \cdot \mathbb{P}\left(K_{\mathsf{a}} = x + w - k_0\right) + \sum_{k_0=d+1}^{w} \mathbb{P}\left(K_0 = k_0\right) \cdot \mathbb{P}\left(K_{\mathsf{a}} = x + w - d\right) \quad (3.4)$$

From here, we proceed to solve for the best distribution for $K_0, \ldots, K_m$ to minimize (3.1) subject

to constraints (3.2)–(3.4). That is, we specify those constraints, along with

$$\forall i, i', k: \ \mathbb{P}\left(K_i = k\right) = \mathbb{P}\left(K_{i'} = k\right) \tag{3.5}$$

$$\forall i: \ \sum_{k_i=0}^{w} \mathbb{P}\left(K_i = k_i\right) = 1 \tag{3.6}$$

$$\forall i, k_i: \ \mathbb{P}\left(K_i = k_i\right) \geq 0 \tag{3.7}$$

and then solve for each $\mathbb{P}\left(K_i = k_i\right)$ to minimize (3.1).

Unfortunately, solving to minimize (3.1) alone simply results in a distribution that results in no use of the cache at all (e.g., $\mathbb{P}\left(K_i = 0\right) = 1$ for each i). As such, we need to rule out such degenerate and "unfair" cases:

$$\forall i: \ \mathbb{P}\left(K_i < w/(m+1)\right) = 0 \tag{3.8}$$

Also, to encourage cache usage, we counterbalance (3.1) with a second goal that values greater use of the cache. We express this goal as minimizing the earth mover's distance [35] from the distribution that assigns $\mathbb{P}\left(K_i = w\right) = 1$, i.e.,

$$\sum_{k=0}^{w} (w-k) \cdot \mathbb{P}\left(K_0 = k\right) \tag{3.9}$$

As such, the final optimization problem seeks to balance (3.1) and (3.9). Let constant φ denote the maximum (i.e., worst) possible value of (3.1) (i.e., when $\mathbb{P}\left(K_i = w\right) = 1$ for each i) and α denote the maximum (i.e., worst) possible value of (3.9) (i.e., when $\mathbb{P}\left(K_i = 0\right) = 1$ for each i). Then, given a parameter ρ, $0 < \rho < 1$, our optimization computes distributions for $K_0, \ldots, K_m$ so as to minimize u subject to

$$u = \frac{1}{\varphi} \left(\sum_{0 \leq d < d' \leq w} \sum_{x} \left| \mathbb{P}_d\left(X = x\right) - \mathbb{P}_{d'}\left(X = x\right) \right| \right)$$

$$u \geq \frac{1}{\alpha(1+\rho)} \left(\sum_{k=0}^{w} (w-k) \cdot \mathbb{P}\left(K_0 = k\right) \right)$$

and constraints (3.2)–(3.8).

The evaluation in Sec. 3.3.2 empirically characterizes the security that result from setting $\rho = 0.01$ the default setting in CacheBar.

3.2.3 Implementation

Implementation of cacheable queues is processor micro-architecture dependent. Here we focus our attention on Intel x86 processors, which appears to be more vulnerable to Prime+Probe attacks due to their inclusive last-level cache [68]. As x86 architectures only support memory management at the page granularity (e.g., by manipulating the PTEs to cause page faults), CacheBar controls the cacheability of memory blocks at page granularity. CacheBar uses reserved bits in each PTE to manage the cacheability of, and to track accesses to, the physical page to which it points, since a reserved bit set in a PTE induces a page fault upon access to the associated virtual page, for which the backing physical page cannot be retrieved or cached (if it is not already) before the bit is cleared [49, 80]. We hence use the term *domain-cacheable* to refer to a physical page that is "cacheable" in the view of all processes in a particular security domain, which is implemented by modifying all relevant PTEs (to have no reserved bits set) in the processes of that security domain. By definition, a physical page that is domain-cacheable to one container may not necessarily be domain-cacheable to another.

To ensure that no more than k_i memory blocks from all processes in container i can occupy lines in a given cache set, CacheBar ensures that no more than k_i of those processes' physical memory pages, of which contents can be stored in that cache set, are domain-cacheable at any point in time. Physical memory pages of which contents can be stored in the same cache set are said to be of the same *color*, and so to implement this property, CacheBar maintains, per container and per color (rather than per cache set), one cacheable queue, each element of which is a physical memory page that is domain-cacheable in this container. Since the memory blocks in each physical page map to different cache sets, limiting the domain-cacheable pages of a color to k_i also limits the number of cache lines that blocks from these pages can occupy in the same cache set to k_i.

To implement a non-domain-cacheable memory, CacheBar uses one reserved bit, which we denote by `NC`, in all PTEs within the domain mapped to that physical page. As such, accesses to any of these virtual pages will be trapped into the kernel and handled by the page fault handler. Upon detecting page faults of this type, the page fault handler will move the accessed physical page

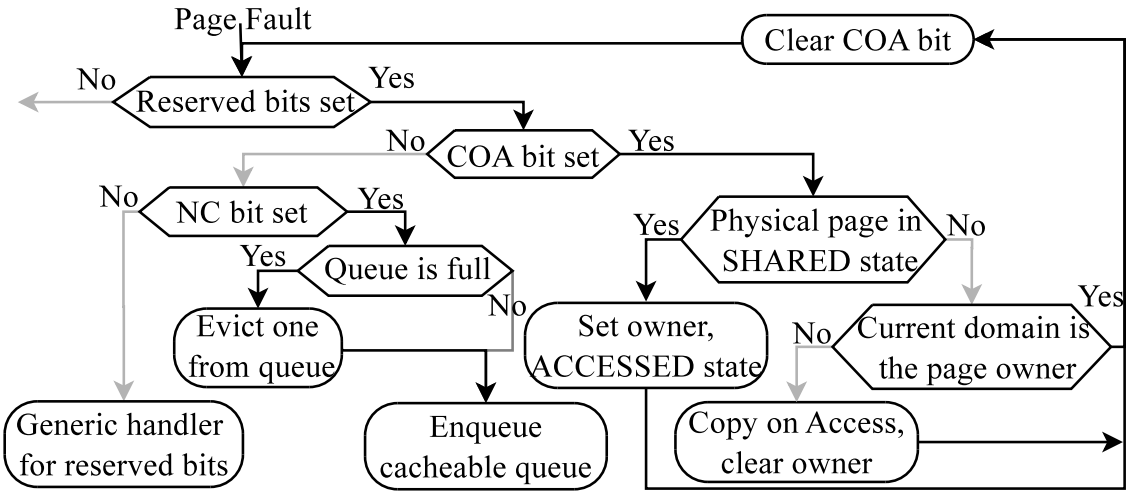


Figure 3.4: Page fault handler for CacheBar.

into the corresponding cacheable queue, clear the `NC` bit in the current PTE[4], and remove a least recently used physical page from the cacheable queue and set the `NC` bits in this domain's PTEs mapped to that page. A physical page removed from the cacheable queue will be flushed out of the cache using `clflush` instructions on all of its memory blocks to ensure that no residue remains in the cache. CacheBar will flush the translation lookaside buffers (TLB) of all processors to ensure the correctness of page cacheabilities every time PTEs are altered. In this way, CacheBar limits the number of domain-cacheable pages of a single color at any time to k_i.

To maintain the LRU property of the cacheable queue, a daemon periodically re-sorts the queue in descending order of recent access count. Specifically, the daemon traverses the domain's page table entries mapped to the physical frame within that domain's queue and counts the number having their ACCESSED bit set, after which it clears these ACCESSED bits. It then orders the physical pages in the cacheable queue by this count (see Fig. 3.3). In our present implementation, this daemon is the same daemon that resets pages from the ACCESSED state to SHARED state (see Sec. 3.1), which already checks and resets the ACCESSED bits in copies' PTEs. Again, this daemon runs every $\Delta T_{\mathsf{acc}} = 1$s seconds in our implementation. This daemon also performs the task of resetting k_i for each security domain i, each time it runs.

[4]We avoid the overhead of traversing all PTEs in the container that map to this physical page. Access to those virtual pages will trigger page faults to make these updates without altering the cacheable queue.

Interacting with copy-on-access The cacheable queues work closely with the copy-on-access mechanisms. In particular, as both the COA and `NC` bits may trigger a page fault upon page accesses, the page handler logic must incorporate both (see Fig. 3.4). First, a page fault is handled as normal unless it is due to one of the reserved bits set in the PTE. As CacheBar is the only source of reserved bits, it takes over page fault handling from this point. CacheBar first checks the COA bit in the PTE. If it is set, the corresponding physical page is either SHARED, in which case it will be transitioned to ACCESSED, or ACCESSED, in which case it will be copied and transitioned to either SHARED or EXCLUSIVE. CacheBar then clears the COA bit and, if no other reserved bits are set, the fault handler returns. Otherwise, if the `NC` bit is set, the associated physical page is not in the cacheable queue for its domain, and so CacheBar enqueues the page and, if the queue is full, removes the least-recently-used page from the queue. If the `NC` bit is clear, this page fault is caused by unknown reasons and CacheBar turns control over to the generic handler for reserved bits.

3.3 Security Evaluation

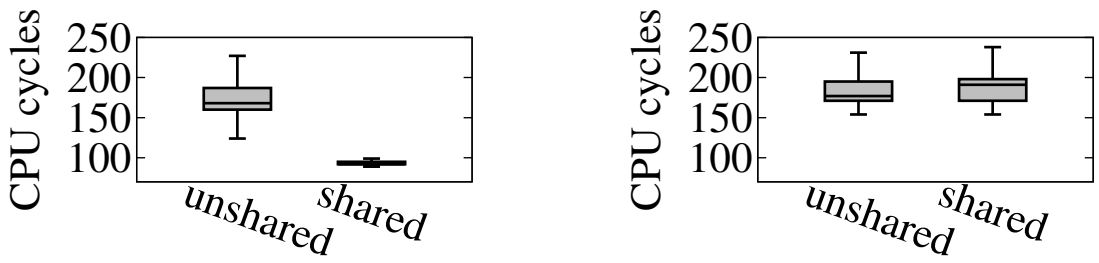
In this section, We empirically evaluated the effectiveness of CacheBar in defending against both Flush+Reload and Prime+Probe attacks.

Our testbed is a rack mounted DELL server equipped with two 2.67GHz Intel Xeon 5550 processors. Each processor contains 4 physical cores (hyperthreading disabled) sharing an 8MB last-level cache (L3). Each core has a 32KB L1 data and instruction cache and a 256KB L2 unified cache. The rack server is equipped with 128GB DRAM and 1000MB NIC connected to a 1000MB ethernet.

We implemented CacheBar as a kernel extension for Linux kernel 3.13.11.6 that runs Ubuntu 14.04 server edition. We set up containers using Docker 1.7.1.

3.3.1 Flush+Reload attacks

We constructed a Flush+Reload covert channel between sender and receiver processes, which were isolated in different containers. Both the sender and receiver were linked to a shared library, `libcrypto.so.1.0.0`, and were pinned to run on different cores of the same socket, thus sharing the same last-level cache. The sender ran in a loop, repeatedly accessing one memory location (the beginning address of function `AES_decrypt()`). The receiver executed Flush+Reload attacks on the same memory address, by first FLUSHing the memory block out of the shared `LLC` with an



(a) CacheBar disabled (b) CacheBar enabled

Figure 3.5: Reload timings in Flush+Reload attacks on a shared address vs. on an unshared address

`clflush` instruction and then Reloading the block by accessing it directly while measuring the access latency. The interval between Flush and Reload was set to 2500 cycles. The experiment was run for 500,000 Flush+Reload trials. We then repeated this experiment with the sender accessing an unshared address, to form a baseline.

Fig. 3.5(a) shows the results of this experiment, when run over unmodified Linux. The three horizontal lines forming the "box" in each boxplot represents the first, second (median), and third quartiles of the Flush+Reload measurements; whiskers extend to cover all points that lie within 1.5× the interquartile range. As can be seen in this figure, the times observed by the receiver to Reload the shared address were clearly separable from the times to Reload the unshared address, over unmodified Linux. With CacheBar enabled, however, these measurements are no longer separable (Fig. 3.5(b)). Certain corner cases are not represented in Fig. 3.5. For example, we found it extremely difficult to conduct experiments to capture the corner cases where Flush and Reload takes place right before and after physical page mergers, as described in Sec. 3.3.1(a) . As such, we rely on our manual inspection of the implementation in these cases to check correctness and argue these corner cases are very difficult to exploit in practice.

3.3.1(a) Model Checking Noninterference Copy-on-access is intuitively secure by design, as no two security domains may access the same physical page at the same time, rendering a general Flush +Reload attack seemingly impossible, as demonstrated in previous section. To show security formally, we subjected our design to model checking in order to ensure that copy-on-access is secure against Flush+Reload attacks at every execution point. Model checking is an approach to formally verify a specification of a finite-state concurrent system expressed as temporal

logic formulas, by traversing the finite-state machine defined by the model. In our study, we used the `Spin` model checker, which offers efficient ways to model concurrent systems and verify temporal logic specifications.

System modeling We model a physical page in Fig. 3.1 using a byte variable in the PROMELA programming language, and two physical pages as an array of two such variables, named `pages`. We model two security domains (e.g., containers), an attacker domain and a victim domain, as two processes in PROMELA. Each process maps a virtual page, `virt`, to one of the physical pages. The virtual page is modeled as an index to the `pages` array; initially `virt` for both the attacker and the victim point to the first physical page (i.e., `virt` is 0). The victim process repeatedly sets `pages[virt]` to 1, simulating a memory access that brings `pages[virt]` into cache. The attacker process FLUSHes the virtual page by assigning 0 to `pages[virt]` and RELOADs it by assigning 1 to `pages[virt]` after testing if it already equals to 1. Both the FLUSH and RELOAD operations are modeled as atomic to simplify the state exploration.

We track the state and owner of the first physical page using another two variables, `state` and `owner`. The first page is initially in the SHARED state (`state` is SHARED), and state transitions in Fig. 3.1 are implemented by each process when they access the memory. For example, the RELOAD code snippet run by the attacker is shown in Fig. 3.6. If the attacker has access to the shared page (Line 3), versus an exclusive copy (Line 16), then it simulates an access to the page, which either moves the state of the page to ACCESSED (Line 10) if the state was SHARED (Line 9) or to EXCLUSIVE (Line 14) after making a copy (Line 13) if the state was already ACCESSED and not owned by the attacker (Line 12). Leakage is detected if `pages[virt]` is 1 prior to the attacker setting it as such (Line 19), which the attacker tests in Line 18.

To model the dashed lines in Fig. 3.1, we implemented another process, called *timer*, in PROMELA that periodically transitions the physical page back to SHARED state from ACCESSED state, and periodically with a longer interval, merges the two pages by changing the value of `virt` of each domain back to 0, `owner` to `none`, and `state` to SHARED.

The security specification is stated as a noninterference property. Specifically, as the attacker domain always FLUSHes the memory block (sets `pages[virt]` to 0) before RELOADing it (setting `pages[virt]` to 1), if the noninterference property holds, then the attacker should always find

```
1  atomic {
2  if
3  ::(virt == 0)→
4     if
5     ::(state == UNMAPPED) →
6        assert(0)
7     ::(state == EXCLUSIVE && owner != ATTACKER) →
8        assert(0)
9     ::(state == SHARED) →
10       state = ACCESSED
11       owner = ATTACKER
12    ::(state == ACCESSED && owner != ATTACKER) →
13       virt = 1  /* copy-on-access */
14       state = EXCLUSIVE
15    fi
16 ::else → skip
17 fi
18 assert(pages[virt] == 0)
19 pages[virt] = 1
20 }
```

Figure 3.6: Code snippet for RELOAD.

`pages[virt]` to be 0 upon RELOADing the page. The model checker checks for violation of this property.

Automated verification We checked the model using `Spin`. Interestingly, our first checking attempt suggested that the state transitions may leak information to a FLUSH+RELOAD attacker. The leaks were caused by the *timer* process that periodically transitions the model to a SHARED state. After inspecting the design and implementation, we found that there were two situations that may cause information leaks. In the first case, when the timer transitions the state machine to the SHARED state from the ACCESSED state, if the prior owner of the page was the victim and the attacker reloaded the memory right after the transition, the attacker may learn one bit of information. In the second case, when the physical page was merged with its copy, if the owner of the page was the victim before the page became SHARED, the attacker may reload it and again learn one bit of information. Since in our implementation of CACHEBAR, these two state transitions are triggered if the page (or its copy) has not been accessed for a while (roughly ΔT_{acc} and ΔT_{copy} seconds, respectively), the information leakage bandwidth due to each would be approximately $1/\Delta T_{\mathsf{acc}}$ bits per page per second or $1/\Delta T_{\mathsf{copy}}$ bits per page per second, respectively.

We improved our CACHEBAR implementation to prevent this leakage by enforcing `LLC` flushes (as described in Sec. 3.1.2) upon these two periodic state transitions. We adapted our model accordingly to reflect such changes by adding one more instruction to assign `pages[0]` to be 0 right after the two *timer*-induced state transitions. Model checking this refined model revealed no

further information leakage.

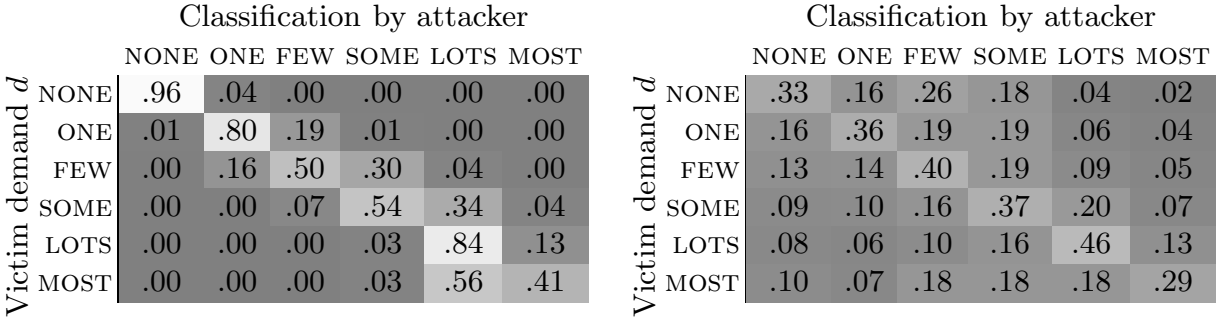
3.3.2 Prime+Probe attacks

We evaluated the effectiveness of CacheBar against Prime+Probe attacks by measuring its ability to interfere with a simulated attack. Because the machine architecture on which we performed these tests had a w-way `LLC` with $w = 16$, we limited our experiments to only a single attacker container (i.e., $m = 1$), but an architecture with a larger w could accommodate more.[5]

In our simulation, a process in the attacker container repeatedly performed Prime+Probe attacks on a specific cache set, while a process in a victim container accessed data that were retrieved into the same cache set at the rate of d accesses per attacker Prime+Probe interval. The cache lines available to the victim container and attacker container, i.e., k_{v} and k_{a} respectively, were fixed in each experiment. The calculations in Sec. 3.2.2 implied that k_{v} and k_{a} could take on values from $\{4, 5, 6, \ldots, 14\}$. In each test with fixed k_{v} and k_{a}, we allowed the victim to place a demand of (i.e., retrieve memory blocks to fill) $d \in \{0, 1, 2, ..., 16\}$ cache lines of the cache set undergoing the Prime+Probe attack by the attacker. The attacker's goal was to classify the victim's demand into one of six classes: none $= \{0\}$, one $= \{1\}$, few $= \{2, 3, 4\}$, some $= \{5, 6, 7, 8\}$, lots $= \{9, 10, 11, 12\}$, and most $= \{13, 14, 15, 16\}$.

To make the attack easier, we permitted the attacker to know k_{a}; i.e., the attacker trained a different classifier per value of k_{a}, with knowledge of the demand d per Prime+Probe trial, and then tested against additional trial results to classify unknown victim demands. Specifically, after training a naïve Bayes classifier on 500,000 Prime+Probe trials per $(d, k_{\mathsf{a}}, k_{\mathsf{v}})$ triple, we tested it on another 500,000 trials. To filter out Probe readings due to page faults, excessively large readings were discarded from our evaluation. The tests without CacheBar yielded the confusion matrix in Table 3.7(a), with overall accuracy of 67.5%. In this table, cells with higher numbers have lighter backgrounds, and so the best attacker would be one who achieves white cells along the diagonal and dark-gray cells elsewhere. As can be seen there, classification by the attacker was very accurate for d falling into none, one, or lots; e.g., $d = 1$ resulted in a classification of one

[5]For example, on an Itanium 2 processor with a 64-way `LLC`, CacheBar could accommodate $m = 3$ or larger. That said, we are unaware of prior works that have successfully conducted Prime+Probe attacks from multiple colluding attackers, which would itself face numerous challenges (e.g., coordinating Probes by multiple processes).



(a) Without CacheBar (b) With CacheBar

Figure 3.7: Confusion matrix of naïve Bayes classifier

with probability of 0.80. Other demands had lower accuracy, but were almost always classified into adjacent classes; i.e., *every* class of victim demand was classified correctly or as an adjacent class (e.g., $d \in$ FEW was classified as ONE, FEW, or SOME) at least 96% of the time.

In contrast, Fig. 3.7(b) shows the confusion matrix for a naïve Bayes classifier trained and tested using Prime+Probe trials conducted with CacheBar enabled. Specifically, these values were calculated using

$$\mathbb{P}\big(\mathsf{class} = c \;\big|\; d \in c'\big) = \sum_{4 \le k_\mathsf{a}, k_\mathsf{v} \le 14} \left(\begin{array}{r} \mathbb{P}\Big(\mathsf{class} = c \;\Big|\; d \in c' \wedge K_\mathsf{v} = k_\mathsf{v} \wedge \;\; K_\mathsf{a} = k_\mathsf{a}\Big) \\ \cdot\; \mathbb{P}\left(K_\mathsf{a} = k_\mathsf{a}\right) \cdot \mathbb{P}\left(K_\mathsf{v} = k_\mathsf{v}\right) \end{array} \right)$$

where class denotes the classification obtained by the adversary using the naïve Bayes classifier; $c, c' \in \{$NONE, ONE, FEW, SOME, LOTS, MOST$\}$; and $\mathbb{P}\left(K_\mathsf{a} = k_\mathsf{a}\right)$ and $\mathbb{P}\left(K_\mathsf{v} = k_\mathsf{v}\right)$ are calculated as described in Sec. 3.2.2. The factor
$\mathbb{P}\big(\mathsf{class} = c \;\big|\; d \in c' \wedge K_\mathsf{v} = k_\mathsf{v} \wedge K_\mathsf{a} = k_\mathsf{a}\big)$ was measured empirically. Though space limits preclude reporting the full class confusion matrix for each k_v, k_a pair, the accuracy of the naïve Bayes classifier per k_v, k_a pair, averaged over all classes c, is shown in Fig. 3.8. As in Fig. 3.7, cells with larger values in Fig. 3.8 are more lightly colored, though in this case, the diagonal has no particular significance. The design intuitively assumes when the attacker and victim are each limited to fewer lines in the cache set (i.e., small values of k_a and k_v, in the upper left-hand corner of Fig. 3.8) the accuracy of the attacker will suffer, whereas when the attacker and victim are permitted to use more lines of the cache (i.e., in the lower right-hand corner) the attacker's accuracy would improve. Fig. 3.8 supports these general trends.

k_a \ k_v	4	5	6	7	8	9	10	11	12	13	14
4	.18	.17	.17	.17	.17	.17	.17	.17	.36	.22	.33
5	.19	.17	.30	.32	.27	.27	.20	.26	.33	.46	.39
6	.17	.31	.24	.18	.21	.17	.20	.27	.43	.39	.41
7	.17	.33	.22	.22	.19	.31	.33	.33	.46	.48	.54
8	.33	.35	.32	.23	.43	.37	.43	.42	.32	.38	.49
9	.20	.26	.31	.28	.44	.38	.34	.34	.46	.39	.56
10	.41	.31	.27	.35	.50	.55	.53	.31	.53	.50	.62
11	.45	.45	.40	.45	.47	.54	.54	.57	.67	.50	.50
12	.55	.50	.59	.63	.49	.48	.54	.49	.56	.58	.57
13	.55	.53	.68	.68	.54	.65	.52	.56	.57	.66	.66
14	.53	.56	.45	.65	.46	.62	.48	.68	.55	.57	.53

Figure 3.8: Accuracy per values of k_v and k_a

Fig. 3.7(b) shows that CacheBar substantially degrades the adversary's classification accuracy, which overall is only 33%. Moreover, the adversary is not only wrong more often, but is also often "more wrong" in those cases. That is, whereas in Fig. 3.7(a) shows that each class of victim demand was classified as that demand or an adjacent demand at least 96% of the time, this property no longer holds true in Fig. 3.7(b). Indeed, the attacker's *best* case in this regard is classifying victim demand LOTS, which it classifies as SOME, LOTS, or MOST 75% of the time. In the case of a victim demand of MOST, this number is only 47%.

3.4 Summary

This chapter presented two techniques to defend against side-channel attacks via LLCs, namely (i) copy-on-access for physical pages shared among multiple security domains, to interfere with Flush+Reload attacks, and (ii) cacheability management for pages to limit the number of cache lines per cache set that an adversary can occupy simultaneously, to mitigate Prime+Probe attacks. Using formal analysis (model checking for copy-on-access, and probabilistic modeling for cacheability management), we developed designs that mitigate side-channel attacks in our empirical evaluations. We also learned a lesson that the experiment-based leakage measure covers fewer leaks than a static analysis because its empirical data is limited to concrete attacks.

CHAPTER 4: STATIC ANALYSIS OF QUANTITATIVE NONINTERFERENCE[1]

The quantitative noninterference evaluation for cache mitigation in the previous Chapter 3 is attack-specific. In this chapter, we propose a static method to measure interference in software using static analysis before it happens.

Our intuition draws from *noninterference* [44], which informally is achieved when the attacker-controlled inputs and attacker-observable outputs are unchanged by the value of a secret input that should not "interfere" with what the attacker can observe. In principle, for any secret S, we could build pairs of all attacker-controlled inputs C with the attacker-observable O outputs that can possibly result from assignments I(ivars) – we'll call these the $\langle \mathsf{C}, \mathsf{O} \rangle$ pairs for S. If secrets S and S' have different sets of $\langle \mathsf{C}, \mathsf{O} \rangle$ pairs then there will be inputs that reveal interference. Unfortunately, for complex procedures, it is impractical to enumerate all $\langle \mathsf{C}, \mathsf{O} \rangle$ pairs for all possible secrets, so previous explorations based on similar enumerating have been limited (See Sec. 2.3).

By leveraging techniques from *approximate model counting* [15], we show how to scalably estimate the number of $\langle \mathsf{C}, \mathsf{O} \rangle$ pairs to a desired accuracy and confidence and—perhaps more to the point—the number of $\langle \mathsf{C}, \mathsf{O} \rangle$ pairs that are consistent with one or both of two disjoint spaces of secret values. Finding two spaces of secret values for which these counts suggest pairs consistent with one but not both then reveals interference. Moreover, we will demonstrate the need to examine samples of secrets of varying sizes, and show that small samples provide a more reliable indication of the *number* of secret values about which information leaks, whereas larger samples provide more insight into the *amount* of leakage of secret values. In doing so, we develop a powerful framework for interference detection and assessment with the following strengths:

- The error in our assessment of a reported interference can be reduced, arbitrarily close to zero in the limit, through greater computational investment. Specifically, by increasing the accuracy

[1]This chapter is excerpted from previously published work [100] coauthored with Ziyun Qian, Michael K. Reiter, and Yinqian Zhang.

and confidence with which the number of pairs of $\langle \mathsf{C}, \mathsf{O} \rangle$ consistent with sampled secrets are estimated, and by increasing the number and variety of samples tested, the interference assessment quantifiably improves.

- Our framework supports the derivation of values from its estimates that separately provide insight into the *number* of secret values about which information leaks, and the *amount* of leakage about those secrets. Within the context of particular applications, one type of leakage might be more important than the other.
- Even for nondeterministic applications, our framework provides a robust assessment of noninterference, by accounting for the nondeterministic factors (e.g., procedure inputs I other than the secrets or attacker-controlled values).

We demonstrate our tool through its application in numerous scenarios. We first apply it to selected, artificially small examples (microbenchmarks) to demonstrate its features. Then, we apply it to assess leakage in several real-world examples.

- We apply our tool to detect leakage of web search query strings submitted to the `Sphinx` web server on the basis of auto-complete response sizes returned to the client (i.e., even if the query and response contents themselves are encrypted) [18]. We also leverage our tool to evaluate the impact of various mitigation strategies on this leak, e.g., showing that based on the contents of the searchable database, some seemingly stronger defenses offer little additional protection over seemingly weaker ones.
- We use our tool to demonstrate the vulnerability leveraged in Compression Ratio Info-leak Made Easy (`CRIME`) attacks [53], specifically that adaptive compression algorithms provide opportunities for an attacker to test guesses about secrets that he cannot observe, if he can instead observe the length of compressed strings containing both the secret and his guess. This case study demonstrates the ability of our technique to effectively account for attacker-controlled inputs, in contrast to many prior techniques (see Sec. 2.3). Specifically, we apply our tool to both `Gzip` and the fixed-dictionary compression library `Smaz` to illustrate that they both leak information about secrets to the adversary, but that `Gzip` leaks more information as the number of adversary-controlled executions grows.
- We apply our tool to illustrate the tendency of Linux to leak TCP-session sequence numbers to an off-path attacker [72, 79]. This is perhaps the most complex of the examples we consider, and

again illustrates the power of accounting for attacker-controlled variables. Moreover, we evaluate two plausible defenses against this attack, one a hypothetical patch to Linux that we propose, and another being simply to disable use of information that is central to the leak.

This chapter first presents the methodology for interference measurement in Sec. 4.1. The implementation of our tool is described in Sec. 4.2. We then use microbenchmarks in Sec. 4.3 to demonstrate features of our approach, and apply our tool to real-world codebases in Sec. 4.4. Some limitations of our approach are discussed in Sec. 4.5.

4.1 Quantitative Noninterference

To measure the leakage about 'secret' from O, under the adversary's chosen C, we consider the set Y_s of pairs $\langle \mathsf{C}, \mathsf{O} \rangle$ that are consistent with S('secret'):

$$X_s = \left\{ \langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle \;\middle|\; \Pi_{proc}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{O}) \wedge \mathsf{S}(\text{'secret'}) = s \right\}$$
$$Y_s = \{ \langle \mathsf{C}, \mathsf{O} \rangle \mid \exists \mathsf{I} : \langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle \in X_s \}$$

Y_s is an indicator of how s influences the possible view of the adversary. For example, if O is independent of 'secret' and so leaks nothing about the value of 'secret', regardless of how the adversary chooses C, then $Y_s = Y_{s'}$ for any $s, s' \in \mathbb{S}$. To generalize from this example, let $Y_S = \bigcup_{s \in S} Y_s$ and then consider the Jaccard distance of Y_S and $Y_{S'}$ for any two disjoint sets $S, S' \subseteq \mathbb{S}$:

$$J(S, S') = \frac{\left|(Y_S \setminus Y_{S'}) \cup (Y_{S'} \setminus Y_S)\right|}{\left|Y_S \cup Y_{S'}\right|} = 1 - \frac{\left|Y_S \cap Y_{S'}\right|}{\left|Y_S \cup Y_{S'}\right|} \tag{4.1}$$

(By convention, $J(S, S') = 0$ if $Y_S = Y_{S'} = \emptyset$.) On the one hand, $J(S, S') = 0$ implies that $Y_S = Y_{S'}$ or, in other words, that an attacker cannot distinguish whether the secret S('secret') is in S or S'. On the other hand, $J(S, S') > 0$ implies there is some $\langle \mathsf{C}, \mathsf{O} \rangle \in (Y_S \setminus Y_{S'}) \cup (Y_{S'} \setminus Y_S)$, and so the attacker can potentially distinguish between 'secret' having a value in S and the case in which it has a value in S'. $J(S, S')$ is an aggregate measure of leakage considering all possible attacker-controlled input values, instead of a worst-case measure of interference caused by specific attacker-controlled inputs.

Unfortunately, it is generally infeasible to compute $J(S, S')$ for every disjoint pair $S, S' \subseteq \mathbb{S}$, or

even when S, S' are restricted to being singleton sets. We can, however, estimate

$$J_n = \underset{\substack{S,S' : |S| = |S'| = n \\ \wedge\ S \cap S' = \emptyset}}{\text{avg}} J(S, S') \tag{4.2}$$

to a high level of confidence by sampling disjoint sets S, S' of size n (or of expected size n, as we will discuss in Sec. 4.2.2) at random and computing $J(S, S')$ for each.

Jaccard distance is not the only choice to measure the degree of dissimilarity between sets Y_S and $Y_{S'}$ for S and S'. Various similarity definitions are possible between sets, including SørensenDice index (i.e., $\frac{2\left|Y_S \bigcap Y_{S'}\right|}{|Y_S|+\left|Y_{S'}\right|}$), Tversky index (i.e., a parameterized generalization of the Jaccard similarity and the Sørensen similarity), overlap coefficient (i.e., $\frac{\left|Y_S \bigcap Y_{S'}\right|}{min(|Y_S|,\left|Y_{S'}\right|)}$), etc. Their corresponding distance metrics are also possible to measure the interference. In this dissertation, we will use the Jaccard distance to measure the interference between two secret sets.

4.1.1 The need to vary n

Consider an idealized situation in which a procedure leaks the equivalence class into which S('secret') falls, among a set of c "small" equivalence classes $\mathbb{C}_1, \ldots \mathbb{C}_c$ of equal size w. If $\mathbb{C} = \bigcup_{i=1}^{c} \mathbb{C}_i$, then the remaining elements $\mathbb{C}_0 = \mathbb{S} \setminus \mathbb{C}$ form another, "large" equivalence class ($w < |\mathbb{C}_0|$). Let $C_S^{\mathsf{sm}} \subseteq \{\mathbb{C}_1, \ldots, \mathbb{C}_c\}$ denote the small equivalence classes of which S contains elements and $C_S^{\mathsf{lg}} \subseteq \{\mathbb{C}_0\}$ indicate whether S contains representatives of $\mathbb{C}_0$ (in which case $C_S^{\mathsf{lg}} = \{\mathbb{C}_0\}$) or not (in which case $C_S^{\mathsf{lg}} = \{\}$). For simplicity, we assume below that $|Y_{\mathbb{C}_i}|$ is the same for each $i \in \{0, 1, \ldots, c\}$.

For the rest of this discussion, we treat the selection of $s \in S$ and $s \in S'$ as the selection, with

replacement, of $\mathbb{C}_i \ni s$.[2] Then, $\mathbb{E}\left(\left|C_S^{\mathsf{sm}}\right|\right) = c\left(1 - \left(1 - \frac{w}{|\mathbb{S}|}\right)^n\right)$,[3] and so

$$\left|C_S^{\mathsf{sm}} \cup C_{S'}^{\mathsf{sm}}\right| = \left|C_{S\cup S'}^{\mathsf{sm}}\right| = c\left(1 - \left(1 - \frac{w}{|\mathbb{S}|}\right)^{2n}\right)$$
$$\left|C_S^{\mathsf{sm}}\right| + \left|C_{S'}^{\mathsf{sm}}\right| = 2c\left(1 - \left(1 - \frac{w}{|\mathbb{S}|}\right)^{n}\right)$$

Similarly,

$$\mathbb{E}\left(\left|C_S^{\mathsf{lg}} \cup C_{S'}^{\mathsf{lg}}\right|\right) = 1 - \left(\frac{cw}{|\mathbb{S}|}\right)^{2n}$$
$$\mathbb{E}\left(\left|C_S^{\mathsf{lg}}\right| + \left|C_{S'}^{\mathsf{lg}}\right|\right) = 2\left(1 - \left(\frac{cw}{|\mathbb{S}|}\right)^{n}\right)$$

and so

$$\mathbb{E}\left(\left|C_S \cup C_{S'}\right|\right) = c\left(1 - \left(1 - \frac{w}{|\mathbb{S}|}\right)^{2n}\right) + 1 - \left(\frac{cw}{|\mathbb{S}|}\right)^{2n} \tag{4.3}$$
$$\mathbb{E}\left(|C_S| + \left|C_{S'}\right|\right) = 2\left(c\left(1 - \left(1 - \frac{w}{|\mathbb{S}|}\right)^{n}\right) + 1 - \left(\frac{cw}{|\mathbb{S}|}\right)^{n}\right) \tag{4.4}$$

Since $J_n = 1 - \frac{\left|C_S \cap C_{S'}\right|}{\left|C_S \cup C_{S'}\right|} = 2 - \frac{|C_S| + \left|C_{S'}\right|}{\left|C_S \cup C_{S'}\right|}$, we estimate $\mathbb{E}\left(J_n\right) \approx 2 - \frac{\mathbb{E}\left(|C_S| + \left|C_{S'}\right|\right)}{\mathbb{E}\left(\left|C_S \cup C_{S'}\right|\right)}$, using (4.4) and (4.3) for the numerator and denominator, respectively.

- First suppose n is small or, specifically, that $\frac{2nw}{|\mathbb{S}|} \ll 1$. Then, we can apply the binomial approximation $\left(1 - \frac{w}{|\mathbb{S}|}\right)^n \approx 1 - \frac{nw}{|\mathbb{S}|}$ to (4.4) and $\left(1 - \frac{w}{|\mathbb{S}|}\right)^{2n} \approx 1 - \frac{2nw}{|\mathbb{S}|}$ to (4.3) to conclude

$$\mathbb{E}\left(J_n\right) \approx 2 - \frac{\frac{2ncw}{|\mathbb{S}|} + 2 - 2\left(\frac{cw}{|\mathbb{S}|}\right)^n}{\frac{2ncw}{|\mathbb{S}|} + 1 - \left(\frac{cw}{|\mathbb{S}|}\right)^{2n}} \tag{4.5}$$

  Thus, when n is small, $\mathbb{E}\left(J_n\right)$ is sensitive to the number of secrets $cw = |\mathbb{C}|$ about which there is substantial leakage, but is insensitive to c and w individually, i.e., to the *amount* of leakage

[2] In reality, each $\mathbb{C}_i$ can be selected only w times in the drawing of S and S', since S and S' do not intersect. This dependence should not affect our estimates much, however, provided that w is not too small or n is small enough.

[3] Let $X_i = 1$ if class $\mathbb{C}_i \in C_S^{\mathsf{sm}}$ and $X_i = 0$ otherwise. Then, $\mathbb{P}_{X_i=0}\left(=\right)\left(1 - w/\left|\mathbb{S}\right|\right)^n$ and so $\mathbb{P}_{X_i=1}\left(=\right) 1 - \left(1 - w/\left|\mathbb{S}\right|\right)^n$. So, $\mathbb{E}\left(\left|C_S^{\mathsf{sm}}\right|\right) = \sum_{i=1}^{c} \mathbb{E}\left(X_i\right) = \sum_{i=1}^{c} \mathbb{P}_{X_i=1}\left(=\right) c\left(1 - \left(1 - \frac{w}{|\mathbb{S}|}\right)^n\right)$.

about those secrets. As such, small n yields a measure J_n that best indicates the *number* of secrets about which information leaks.

- Now suppose n is large, such that $\left(\frac{cw}{|\mathbb{S}|}\right)^n \approx 0$. Then,

$$\mathbb{E}\left(J_n\right) \approx 2 - \frac{2\left(c\left(1-\left(1-\frac{w}{|\mathbb{S}|}\right)^n\right)+1\right)}{c\left(1-\left(1-\frac{w}{|\mathbb{S}|}\right)^{2n}\right)+1} \tag{4.6}$$

  That is, J_n is sensitive to c and w individually when n is large. In this sense, we say that J_n for large n is a better indicator for the *amount* of leakage about secrets.

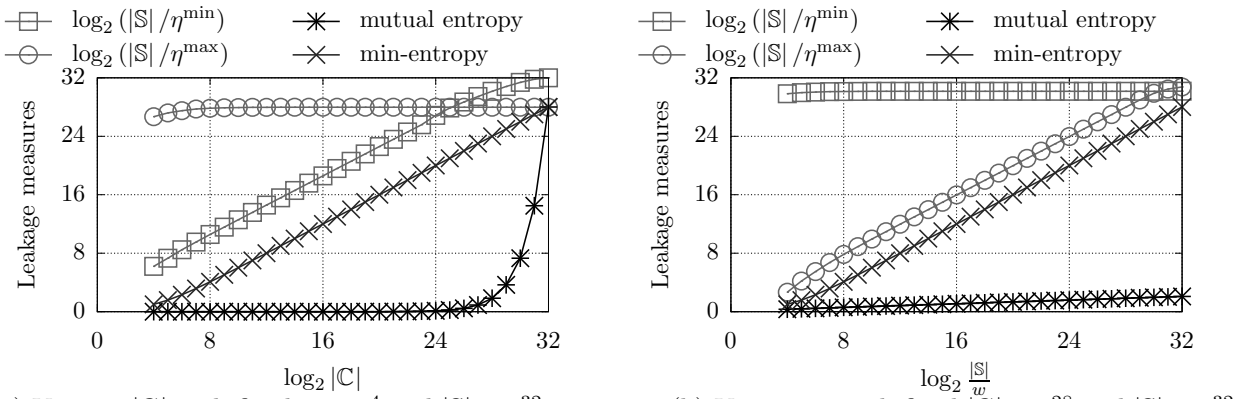
Again, the above model is idealized; leakage from real procedures can be far more complex. Still, this discussion provides insight into the utility of J_n and how it should be used. When n is small, (4.5) grows as $cw = |\mathbb{C}|$ grows, and for any threshold $t \in [0,1]$ indicating "substantial" leakage, the smallest n for which $J_n \geq t$ shrinks. This smallest n is thus a reflection of $|\mathbb{C}|$, i.e., of the number of secrets about which information leaks. When n is large and for a fixed cw, (4.6) grows as w shrinks,[4] and for any threshold $t \in [0,1]$ indicating "substantial" leakage, the largest n for which $J_n \geq t$ grows. This largest n is thus a reflection of w, i.e., of the amount of leakage about those secrets. It is therefore natural to examine both $\min\{n|J_n \geq t\}$ and $\max\{n|J_n \geq t\}$. To define measures using these values that fall within $[0,1]$ and for which larger values indicate more leakage (as with J_n), we define

$$\eta_t^{\min} = \begin{cases} 0 & \text{if } t > J^{\max} \\ 1/\min\{n \mid J_n \geq t\} & \text{otherwise} \end{cases}$$

$$\eta_t^{\max} = \begin{cases} 0 & \text{if } t > J^{\max} \\ \frac{1}{|\mathbb{S}|/2}\max\{n \mid J_n \geq t\} & \text{otherwise} \end{cases}$$

Here, $J^{\max} = \max_{n'} J_{n'}$, and so the $t > J^{\max}$ cases accommodate t values larger than J_n ever reaches. Finally, rather than select a t to define "substantial" leakage, we simply take the average

[4] For example, $\left(1-\frac{w}{|\mathbb{S}|}\right)^n < \frac{1}{2}$ is sufficient to ensure this.



(a) Varying $|\mathbb{C}|$ with fixed $w = 2^4$ and $|\mathbb{S}| = 2^{32}$ (b) Varying w with fixed $|\mathbb{C}| = 2^{28}$ and $|\mathbb{S}| = 2^{32}$

Figure 4.1: Relating $\eta^{\min}$ and $\eta^{\max}$ to min-entropy and mutual entropy, for the idealized model of leakage explored in Sec. 4.1.1

values of $\eta_t^{\min}$ and $\eta_t^{\max}$ over $t \in [0, 1]$ as our final measures:

$$\eta^{\min} = \int_0^1 \eta_t^{\min} dt \qquad \eta^{\max} = \int_0^1 \eta_t^{\max} dt \tag{4.7}$$

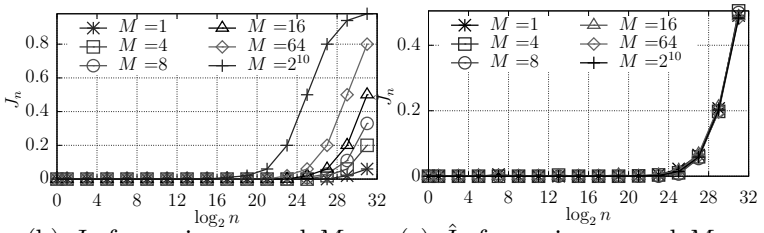
The numbers we report in this chapter are discrete approximations to these values via numerical integration with a fixed subinterval width of 0.01.

Roughly speaking, a larger value for $\eta^{\min}$ suggests that information leaks from the procedure for more secret values, and a larger value for $\eta^{\max}$ suggests that more information leaks from the procedure about secret values.[5] To relate these measures to another used previously in the QIF literature, namely min-entropy (e.g., [87, 31]), in Fig. 4.1 we show $\eta^{\min}$ and $\eta^{\max}$ in comparison to the min-entropy of S('secret'), for our idealized setting above. Fig. 4.1(a) shows that $\eta^{\min}$ reflects the growth of $|\mathbb{C}|$ just as min-entropy can, and similarly, Fig. 4.1(b) shows that $\eta^{\max}$ reflects changes in w like min-entropy can. However, min-entropy does not distinguish between these types of leakage. Mutual entropy (e.g., [23, 59, 69]) also reflects increasing leakage as $|\mathbb{C}|$ grows in Fig. 4.1(a) and as w shrinks in Fig. 4.1(b), though its sensitivity to these effects is limited,

[5]While these rules of thumb are accurate when J_n has no valley, they are less reliable when it does. In such cases, a more reliable understanding can be obtained by examining the graph of J_n directly, or at least by computing a separate $\eta^{\min}$ and $\eta^{\max}$ for each valley-free segment of J_n. Here, by "valley" we mean values n, n' where $n < n'$, $J_n > J_{n+1}$, $J_{n'} < J_{n'+1}$, and $J_{n''} = J_{n''+1}$ for each $n'' \in [n+1, n'-1]$. We have not encountered J_n curves with valleys in practice, and so do not discuss them further here.

```
proc (C, I, S)
    if (S('secret') = C('test'))
        O('result') ← rand() mod M
    else
        O('result') ← M + (rand() mod 16)
    return O
```

(a) Procedure



(b) J_n for various n and M (c) $\hat{J}_n$ for various n and M

Figure 4.2: An example showing limitations of J on procedures with randomness and improvements offered by $\hat{J}$ (see Sec. 4.1.2)

particularly that of increasing $|\mathbb{C}|$, until $|\mathbb{C}|$ becomes quite large (Fig. 4.1(a)).

4.1.2 Procedures with other inputs

The measures J_n, $\eta^{\min}$, and $\eta^{\max}$ are appropriate when *proc* is deterministic and leverages no inputs in I. When either of these restrictions are lifted, our approach described so far can be unreliable. We illustrate this in Sec. 4.1.2(a) and then provide an alternative measure in Sec. 4.1.2(b) that is more robust.

4.1.2(a) Limitations of J_n First consider a randomized password checker that receives a secret password S('secret') and a candidate password C('test') and, for some constant $M > 0$, outputs a random value in $[0, M-1]$ if the candidate password is equal to the secret password and random value in $[M, M+16]$ otherwise. Intuitively, the leakage of this procedure should be the same as a deterministic password checker and independent of the value of M. However, as shown in Fig. 4.2, the use of randomness here results in an unintuitive result, since J_n (Fig. 4.2(b)) is sensitive to the value of M. As such, while our detector does accurately detect leakage in this case, it provides less help in comparing the leakage of two randomized implementations.

Another problem may arise when other inputs are allowed in I. Consider the example

```
proc (C, I, S)
    O('result') ← ((S('secret') > C('test')) ? 1 : 0) ⊕ ((I('other') ≤ 0) ? 1 : 0)
    return O
```

Here, the expression "*cond* ? 1 : 0" evaluates to 1 if *cond* is true and 0 otherwise, and "⊕" represents XOR. This procedure indicates that S('secret') > C('test') by returning 0 if I('other') ≤ 0 or by returning 1 if I('other') > 0. Because our technique allows for any value of I('other') consistent

with Π_{proc} when estimating $|Y_S|$, it will compute $J_n = 0$ for any n, suggesting no leakage. However, the only condition under which *proc* in fact leaks no information is if I('other') is non-positive or positive with equal probability from the adversary's perspective.

4.1.2(b) An alternative measure To overcome the limitations of J_n as illustrated above, in this section we propose a leakage measure that is more robust for procedures that employ randomness or inputs in I. For convenience, here we treat all values generated at random within the procedure instead as inputs represented in I; e.g., the first invocation of `rand()` within the procedure is replaced with a reference to, say, I('rand[1]'), the second with I('rand[2]'), and so forth. Intuitively, our measure employs an alternative definition for Y_S that also includes these additional inputs. Specifically, consider the set $\hat{X}_{S,S'}$

$$\check{X}_{S,S'} = X_S \cup X_{S'}$$
$$\hat{X}_{S,S'} = \left\{ \langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle \,\middle|\, \langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle \in \check{X}_{S,S'} \wedge \langle \mathsf{C}, \mathsf{O} \rangle \in Y_S \cap Y_{S'} \right\}$$

of $\langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle$ triples such that not only is $\langle \mathsf{C}, \mathsf{O} \rangle \in Y_S \cap Y_{S'}$ (c.f., the definition of $J(S, S')$ in (4.1)), but also the triple is consistent with some $s \in S$ (i.e., $\langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle \in X_S$ where $X_S = \bigcup_{s \in S} X_s$). By counting such $\langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle$ triples, the various random values (represented in I) become exposed in $\hat{X}_{S,S'}$ and the number of these values for a given $\langle \mathsf{C}, \mathsf{O} \rangle$ pair act as the "weight" of that pair. When $\langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle$ is from the following difference set, an attacker can determine whether the secret is from S or S'.

$$\tilde{X}_{S,S'} = \check{X}_{S,S'} \setminus \hat{X}_{S,S'} \tag{4.8}$$

We adjust the measure to

$$\hat{J}(S, S') = \frac{\left|\tilde{X}_{S,S'}\right|}{\left|\check{X}_{S,S'}\right|} = 1 - \frac{\left|\hat{X}_{S,S'}\right|}{\left|\check{X}_{S,S'}\right|} \tag{4.9}$$

$$\hat{J}_n = \operatorname*{avg}_{\substack{S, S' : |S| = |S'| = n \\ \wedge \; S \cap S' = \emptyset}} \hat{J}(S, S') \tag{4.10}$$

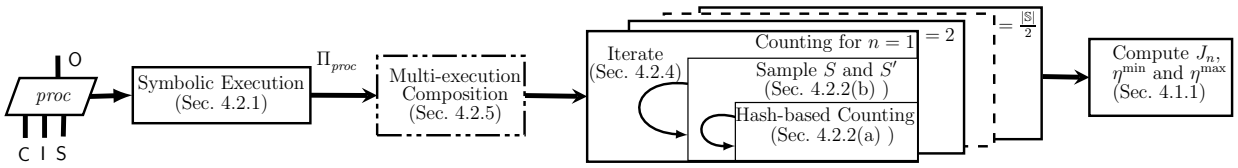


Figure 4.3: Workflow of evaluating leakage, from left to right: label the different types of inputs and outputs; generate postconditions Π_{proc} using symbolic execution; optionally, compose multi-execution constraints; perform model counting for different sizes of n; and generate our leakage measures

Note that if $Vars_{\mathsf{I}} = \emptyset$, then $\hat{J}_n = J_n$ since in this case, $\langle \mathsf{C}, \mathsf{O} \rangle \in Y_S$ if and only if $\langle \mathsf{C}, \mathsf{O}, \emptyset \rangle \in X_S$.

The benefit of $\hat{J}_n$ is that it is far less susceptible to the variability that was demonstrated in Sec. 4.1.2(a) . For example, Fig. 4.2(c) shows that this measure is stable, independent of M. As we will see in subsequent sections, however, it is also considerably costlier to estimate.

When we use $\hat{J}_n$ in place of J_n, we will annotate measures derived from it using similar notation. For example, $\hat{\eta}^{\min}$ denotes $\eta^{\min}$ computed using $\hat{J}_n$ in place of J_n, and similarly for $\hat{\eta}^{\max}$.

4.2 Implementation

In this section, we discuss our implementation for computing the measures discussed in Sec. **??**. Fig. 4.3 shows the overall workflow for doing so. Sec. 4.2.1 illustrates the use of symbolic execution (e.g., [12, 20]) for generating postconditions, with a focus on a particular optimization that proved useful for our case studies in Sec. 4.4. At the core of our implementation is the adaption of hash-based model counting technique that is discussed in Sec. 4.2.2–4.2.4. In Sec. 4.2.5, we present an adaptation for generating logical postconditions for multiple rounds of procedure executions.

4.2.1 From software procedure to logical postcondition

As mentioned in Sec. 4.1, the logical postcondition Π_{proc} represents the relationship between inputs and outputs induced by procedure *proc*. To extract Π_{proc} from *proc*, we apply symbolic execution to *proc*. After marking each input variable (i.e., each parameter in $Vars_{\mathsf{C}}$, $Vars_{\mathsf{I}}$,[6] and $Vars_{\mathsf{S}}$) symbolic before the user-defined entry point, we utilize `KLEE` [12] or `S2E` [20] to explore all feasible execution paths through *proc* that reach a `return`. On each path through *proc*, the symbolic execution engine accumulates a set of constraints among symbolic variables implied by

[6]To model the random input generated from random number generator *rand*() in symbolic execution, we created a symbolic variable per *rand*() function call as its returned value.

the branches taken and assignments computed along that path. These constraints coupled with the assignments for $Vars_{\mathsf{O}}$ defined by our API `make_observable`, as accumulated through the `return` instruction, form the postcondition for the path, and then Π_{proc} is simply the disjunction of the path conditions generated for each execution path.

Symbolic execution can suffer from state explosion, and so we leveraged an optimization in our work to manage this explosion. Specifically, we implemented a searcher to perform *state merging* [61] frequently, wherein the constraints accumulated along two or more execution prefixes ending at the same instruction are disjoined and then simplified to the extent possible (using an SMT solver); execution is then continued from their last instruction, accumulating more constraints into their now-combined constraints. In doing so, these two execution prefixes need only be extended once, versus each being extended separately if no merging occurred.

This optimization dramatically reduced the number of symbolic states managed in one of our case studies in Sec. 4.4.3, improving the speed of extracting Π_{proc} by more than 600×. For this case study, we forced state merging to occur whenever a symbolic state was forked at a symbolic branch. To reduce the complexity of the merged path constraint, however, we avoided merging two path constraints when their expressions for the outputs in O differed or when two path constraints (in conjunctive normal form) had less than half of their conjuncts in common.

To correctly measure the leakage, we assume the postcondition $\Pi_{proc}(\mathsf{C},\mathsf{I},\mathsf{S},\mathsf{O})$ is complete and sound. Completeness means that if $\langle \mathsf{C},\mathsf{I},\mathsf{S},\mathsf{O}\rangle$ is feasible for *proc*, then $\langle \mathsf{C},\mathsf{I},\mathsf{S},\mathsf{O}\rangle$ satisfies $\Pi_{proc}(\mathsf{C},\mathsf{I},\mathsf{S},\mathsf{O})$. Soundness requires that if $\langle \mathsf{C},\mathsf{I},\mathsf{S},\mathsf{O}\rangle$ is infeasible for *proc*, then $\langle \mathsf{C},\mathsf{I},\mathsf{S},\mathsf{O}\rangle$ does not satisfy $\Pi_{proc}(\mathsf{C},\mathsf{I},\mathsf{S},\mathsf{O})$. A well-known limitation of symbolic execution is how to manage unbounded loops, since these can prevent symbolic execution from terminating. In the case studies of Sec. 4.4 we bounded all inputs, which was enough in these case studies to ensure that symbolic execution terminated. Provided that we bound the input parameters sufficiently loosely to encompass all values they can take on in practice, this bounding does not impact the assessment provided by our measures in practice.

Postcondition generation costs are summarized in Fig. 4.1. These computations were performed on a DELL PowerEdge R710 server equipped with two 2.67GHz Intel Xeon 5550 processors and 128GB memory. Each processor includes 4 physical cores and had hyperthreading enabled. As indicated in Fig. 4.1, we experimented with both `KLEE` and `S2E` to generate postconditions, depending

Table 4.1: Postcondition generation times for case studies

Sec.	Procedure	KLEE ×1	KLEE ×16	KLEE ×1, merging	S2E ×16
4.4.1	Auto-complete	2d	12h		
4.4.2	`Gzip`	3d	21h		8h
4.4.2	`Smaz`	2d	18h		6h
4.4.3	v3.18	7d	4d	17m	
4.4.3	v3.18-patched			18m	
4.4.3	v3.18-rmCounter			17m	

on the procedure. In the column headings, a '×1' or '×16' indicates the number of processes across which the computation was divided. To enable multi-process support in `KLEE` (i.e., '×16'), we made a small modification in `KLEE`'s execution engine, to cause it to explore only execution paths starting from a predefined branching prefix. The designation 'merging' indicates the use of the `KLEE` optimization summarized above; as indicated in Fig. 4.1, this optimization was remarkably effective on the Linux TCP implementations discussed in Sec. 4.4.3. `S2E` was configured to utilize its concolic execution capabilities.

4.2.2 Hash-based model counting for J_n

To calculate J_n, we need to estimate $\left|Y_S \cap Y_{S'}\right|$ and $\left|Y_S \cup Y_{S'}\right|$ for randomly selected, disjoint sets S and S' of size n. Since

$$\left|Y_S \cap Y_{S'}\right| = |Y_S| + \left|Y_{S'}\right| - \left|Y_S \cup Y_{S'}\right| \qquad \text{and} \tag{4.11}$$

$$\left|Y_S \cup Y_{S'}\right| = \left|Y_{S \cup S'}\right|, \tag{4.12}$$

to estimate J_n it suffices to estimate $\left|Y_{S''}\right|$ for specified sets S'' (i.e., $S'' = S$, $S'' = S'$, or $S'' = S \cup S'$). In this section, we provide two optimizations for producing such estimates.

4.2.2(a) Estimating $|Y_S|$ Our first optimization is an adaptation of the approximate model counting technique due to Chakraborty et al. [15] (see Sec. 2.4.1).

We estimate $|Y_S|$ through a similar algorithm used in projected model counting, i.e., by iteratively selecting H^b and $p \in \{0,1\}^b$ at random, but apply the hash function only to the C and O values of a satisfying assignment for Π_{proc}. More specifically, we compute the set

$$Z_{S,p} = \left\{ \langle \mathsf{C}, \mathsf{O} \rangle \;\middle|\; \langle \mathsf{C}, \mathsf{O} \rangle \in Y_S \wedge H^b(\langle \mathsf{C}, \mathsf{O} \rangle) = p \right\}$$

That is, $Z_{S,p} \subseteq Y_S$ contains the elements of Y_S whose hash is p. Intuitively, this yields an estimate

$$|Y_S| \approx 2^b \cdot \left|Z_{S,p}\right| \tag{4.13}$$

To reach an estimate of confidence δ, we generate a number of $\langle b, p, \hat{p}\rangle$ triples such that

$$\left|Z_{S,p}\right| \leq \alpha \text{ and } \left|Z_{S,\hat{p}}\right| > \alpha \tag{4.14}$$

where $p \in \{0,1\}^b$, $\hat{p} \in \{0,1\}^{b-1}$, and α is derived from ϵ [15]. Each such triple individually provides an estimate that is within error ϵ with confidence at least 0.78 [15, Lemma 1], and the median of the estimates for all such triples is within error ϵ with confidence that can be increased arbitrarily with more $\langle b, p, \hat{p}\rangle$ such triples. As a special case, if $\left|Z_{S,p}\right| \leq \alpha$ at $b = 0$, then $\left|Z_{S,p}\right|$ is an exact count of $|Y_S|$ since $Z_{S,p} = Y_S$.

4.2.2(b) Sampling S, S' of Expected Size n A second expense of calculating Y_S and $Y_{S'}$ explicitly is in enumerating S and S' themselves, especially if n is large. We can leverage hashing similarly to the method above to avoid enumerating S and S' directly for $n = |\mathbb{S}|/2^b$ for some $b \geq 0$.

Specifically, to estimate J_n for $n = |\mathbb{S}|/2^b$, we select H^b and $p \in \{0,1\}^{b-1}$ at random and, for each such selection, define

$$X_p^0 = \left\{ \langle \mathsf{C}, \mathsf{O}, \mathsf{I}\rangle \;\middle|\; \exists \mathsf{S} : \Pi_{proc}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{O}) \wedge H^b(\mathsf{S}) = p||0 \right\}$$
$$X_p^1 = \left\{ \langle \mathsf{C}, \mathsf{O}, \mathsf{I}\rangle \;\middle|\; \exists \mathsf{S} : \Pi_{proc}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{O}) \wedge H^b(\mathsf{S}) = p||1 \right\}$$
$$X_p = \left\{ \langle \mathsf{C}, \mathsf{O}, \mathsf{I}\rangle \;\middle|\; \exists \mathsf{S} : \Pi_{proc}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{O}) \wedge H^{b-1}(\mathsf{S}) = p \right\}$$

where H^{b-1} denotes the function H^b but dropping the rightmost bit from the output. Then, we

use the sets

$$Y_p^0 = \left\{ \langle \mathsf{C}, \mathsf{O} \rangle \;\middle|\; \exists \mathsf{I} : \langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle \in X_p^0 \right\}$$
$$Y_p^1 = \left\{ \langle \mathsf{C}, \mathsf{O} \rangle \;\middle|\; \exists \mathsf{I} : \langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle \in X_p^1 \right\}$$
$$Y_p = \left\{ \langle \mathsf{C}, \mathsf{O} \rangle \;\middle|\; \exists \mathsf{I} : \langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle \in X_p \right\}$$

in place of Y_S, $Y_{S'}$, and $Y_{S \cup S'}$, respectively, to perform the calculations (4.11)–(4.12). And, of course, the optimization in Sec. 4.2.2(a) can be used in conjunction with this approach, e.g., computing

$$Z_{p,\hat{p}}^0 = \left\{ \langle \mathsf{C}, \mathsf{O} \rangle \;\middle|\; \langle \mathsf{C}, \mathsf{O} \rangle \in Y_p^0 \wedge \hat{H}^{\hat{b}}(\langle \mathsf{C}, \mathsf{O} \rangle) = \hat{p} \right\} \tag{4.15}$$

for a different, random hash function $\hat{H}^{\hat{b}}$ and random prefix $\hat{p} \in \{0,1\}^{\hat{b}}$. We then use the algorithm summarized in Sec. 4.2.2(a) to estimate $\left|Y_p^0\right|$.

Two more points about this algorithm warrant emphasis:

- Because our algorithm explicitly enumerates the contents of each $Z_{p,\hat{p}}^0$ and $Z_{p,\hat{p}}^1$, when leakage is detected (i.e., $J_n > 0$ for some n) these sets can be used to identify $\langle \mathsf{C}, \mathsf{O} \rangle$ pairs that are in $Y_p^0 \setminus Y_p^1$ or $Y_p^1 \setminus Y_p^0$. These examples can guide developers in understanding the reason for the leakage and in mitigating the problem.
- Because the number of secrets with a random length-b hash prefix p is only of *expected* size $n = |\mathbb{S}| / 2^b$, for the rest of the chapter we use a definition of J_n as in (4.2) but weakened so that $|S|$ and $|S'|$ equal n in expectation.

4.2.3 Hash-based model counting for $\hat{J}_n$

The calculations of the previous section require some modifications when we are instead computing $\hat{J}_n$ for $n = |\mathbb{S}| / 2^b$. Similar to the previous section, we can use X_p for $p \in \{0,1\}^{b-1}$ in place of $X_S \cup X_{S'} = X_{S \cup S'}$. However, to estimate $\left|\hat{X}_{S,S'}\right|$ for a random S and S', we need a different approach. Specifically, we calculate $\left|\hat{X}_{S,S'}\right|$ by estimating the size of

$$\hat{X}_p = \left\{ \langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle \;\middle|\; \begin{array}{c} \exists \mathsf{S}, \mathsf{S}', \mathsf{I}' : \;\; \Pi_{proc}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{O}) \;\wedge \Pi_{proc}(\mathsf{C}, \mathsf{I}', \mathsf{S}', \mathsf{O}) \wedge \\ H^b(\mathsf{S}) = p||0 \;\wedge H^b(\mathsf{S}') = p||1 \end{array} \right\}$$

since $\langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle \in \hat{X}_p$ iff $\langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle \in X_p^0$ and $\langle \mathsf{C}, \mathsf{O} \rangle \in Y_p^0 \cap Y_p^1$. This method does come at considerably greater computational cost, however, due to the duplication of the constraints Π_{proc} in the specification of this set. We will demonstrate this in our case studies in Sec. 4.4.

4.2.4 Parameter settings for computing J_n and $\hat{J}_n$

In the hash-based model counting described above, we use the 3-wise independent hash functions suggested by Chakraborty et al. [15], and due to the large number of XOR clauses in the resulting hash constraints, we use `CryptoMiniSAT 5.0` [89] to enumerate the elements of each $Z_{p,\hat{p}}$. To reduce the complexity of the hash constraints, we concretize their constant bits to minimize the independent support [51] before generating XOR clauses. Multiple estimates of the form in (4.13), for various values of b (in (4.13), or respectively $\hat{b}$ in (4.15)), as prescribed by Chakraborty et al., are used to estimate $|Y_p|$. We parameterized this algorithm with error $\epsilon = 0.45$ and confidence either $\delta = 0.99$ in Sec. 4.3 or $\delta = 0.92$ in Sec. 4.4,[7] for which 50 or 5 $\langle b, p, \hat{p} \rangle$ triples satisfying (4.14) sufficed, respectively.

We estimate J_n as the sample mean of $J(S, S')$ for sampled pairs S, S' of expected size n (i.e., defined by a $p \in \{0,1\}^{b-1}$ for $n = |\mathbb{S}|/2^b$). For each n we computed J_n using a number of sampled pairs S, S' equal to the larger of 100 and the minimum needed so that the standard error was within 5% of the sample mean.

In addition, since J_n is only an estimate and so is subject to error and since that error is influential in the calculation of $\eta^{\max}$ or $\eta^{\min}$ especially when n is small, we round any $J_n \leq 0.025$ down to zero when calculating the measures. $\hat{J}_n$ is computed similarly.

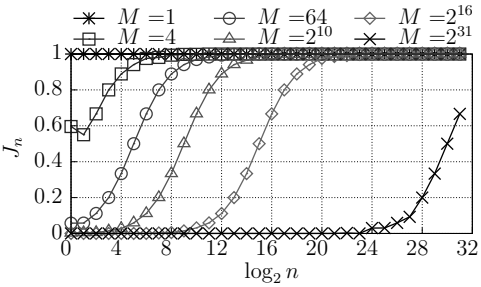
4.2.5 Logical postconditions for multiple procedure executions

In some scenarios it is insightful to observe the behavior of J_n for a procedure *proc* when it is executed multiple times. That is, consider a scenario in which *proc* is executed r times, possibly with relationships among the outputs of one execution and the inputs of another, or simply among

[7] The error bound of Chakroborty et al. is conservative; e.g., the results for 95 benchmarks showed less than 5% error in practice even when using $\epsilon = 0.75$ [15].

```
proc (C, I, S)
    if (S('secret') mod M = 0)
        O('result') ← S('secret')
    else
        O('result') ← 0
    return O
```

(a) Procedure



(b) J_n for different n and M

M	$\log_2 \eta^{\min}$	$\log_2 \eta^{\max}$
1	0	0
4	−0.74	0
64	−4.8	0
2^{10}	−8.8	0
2^{16}	−15	0
2^{31}	−30	−0.67

(c) $\eta^{\min}$ and $\eta^{\max}$ for different M

Figure 4.4: A procedure that leaks about more secrets as M is decreased (see Sec. 4.3.1)

the inputs to different executions. Suppose these executions are denoted

$$\mathsf{O}_1 \leftarrow \mathit{proc}(\mathsf{C}_1, \mathsf{I}_1, \mathsf{S}_1)$$

$$\mathsf{O}_2 \leftarrow \mathit{proc}(\mathsf{C}_2, \mathsf{I}_2, \mathsf{S}_2)$$

$$\ldots$$

$$\mathsf{O}_r \leftarrow \mathit{proc}(\mathsf{C}_r, \mathsf{I}_r, \mathsf{S}_r)$$

and that the postcondition of the j-th invocation in isolation is denoted Π^j_{proc} (i.e., Π^j_{proc} is simply Π_{proc} over the variables represented in C_j, I_j, S_j, and O_j). Then the relationships among inputs and outputs can be described using additional, manually constructed constraints $\Gamma^{1\ldots r}_{proc}$. For example, if the secret input to each execution of *proc* is the same, then $\Gamma^{1\ldots r}_{proc}$ would include the statement that 'secret' has the same value in each execution (i.e., $\mathsf{S}_1(\text{'secret'}) = \mathsf{S}_2(\text{'secret'}) = \ldots = \mathsf{S}_r(\text{'secret'})$). Repeating our analysis for the "procedure" represented by the postcondition

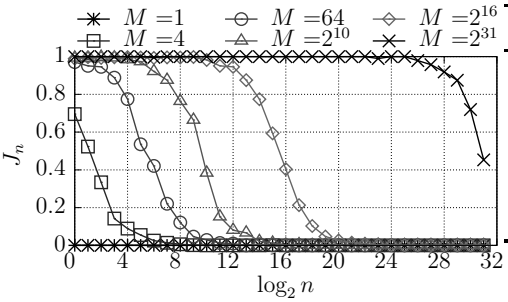
$$\left(\bigwedge_{j=1}^{r} \Pi^j_{proc} \right) \wedge \Gamma^{1\ldots r}_{proc}$$

can reveal leakage that increases as the procedure is executed multiple times. We will see an example in Sec. 4.4.

4.3 Microbenchmark Evaluation

In this section we evaluate our methodology on artificially small examples to illustrate its features.

```
proc (C, I, S)
    O('result') ←
        S('secret') mod M
    return O
```



M	$\log_2 \eta^{\min}$	$\log_2 \eta^{\max}$
1	nan	nan
4	-0.6	-30.1
64	-0.0	-25.5
2^{10}	-0.0	-21.8
2^{16}	-0.0	-15.6
2^{31}	0.0	-0.8

"nan" denotes "not a number," i.e., $\eta^{\min} = 0$ or $\eta^{\max} = 0$

(a) Procedure (b) J_n for different n and M (c)$\eta^{\min}$ and $\eta^{\max}$ for different M

Figure 4.5: A procedure that leaks more about secret values as M is increased (see Sec. 4.3.1)

4.3.1 Leaking more about secret values vs. leaking about more secret values

In Sec. 4.1.1, we showed through an idealized example how a small n is more useful for evaluating the number of secrets about which information leaks, whereas a large n is more useful for evaluating the amount of information leaked about these secrets. Now we will use two simple procedures with a controllable constant M to quantitatively demonstrate the necessity of varying n and the correct usage of $\eta^{\min}$ and $\eta^{\max}$.

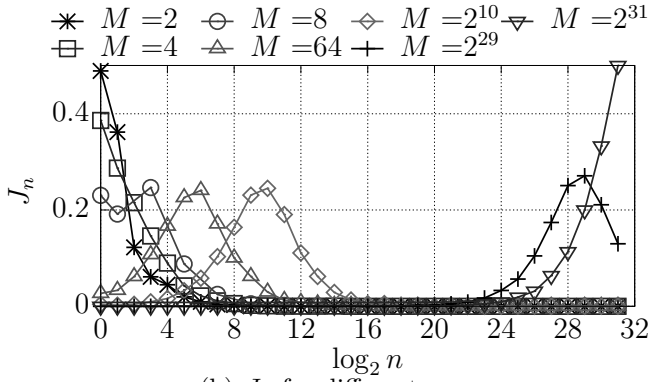
The first procedure, shown in Fig. 4.4(a), returns the secret value if it is divisible by a constant M and returns zero otherwise, where both S('secret') and M are 32-bit integers. This procedure leaks the same amount of information (the whole secret) about a larger number of secret values if M is decreased. The behavior of J_n shown in Fig. 4.4(b) is consistent with this observation. Specifically, different values of M induce curves for J_n that differ primarily in the minimum value of n where J_n is large. This behavior is also seen in the value of $\eta^{\min}$ in Fig. 4.4(c), where $\eta^{\min}$ ranges from $\eta^{\min} \approx 0$ at $M = 2^{31}$ to $\eta^{\min} = 1$ at $M = 1$.

Contrast this case with the procedure shown in Fig. 4.5(a), which returns the residue class of the secret value modulo a constant value M. As such, as M is increased, more information about each secret is leaked. This is demonstrated in Fig. 4.5(b), where the curves for different values of M differ in primarily in the maximum value n at which J_n is large. Similarly, $\eta^{\max}$ ranges from $\eta^{\max} = 0$ at $M = 1$ to $\eta^{\max} \approx 2^{-0.8} \approx 0.57$ at $M = 2^{31}$.

An example that blends these the previous two examples is show in in Fig. 4.6(a); here the procedure returns 1 if S('secret') mod M = C('test') and 0 otherwise, where M is a 32-bit constant. As such, this procedure leaks a lot about a few secret values when M is large, and a little about many secret values when M is small. As shown in the $r = 1$ columns of Fig. 4.6(c), $\eta^{\min}$ and $\eta^{\max}$

```
proc (C, I, S)
    if (S('secret') mod M = C('test'))
        O('result') ← 1
    else
        O('result') ← 0
    return O
```

(a) Procedure



(b) J_n for different n

M	$\log_2 \eta^{\min}$				$\log_2 \eta^{\max}$			
	$r=1$	$r=2$	$r=4$	$r=6$	$r=1$	$r=2$	$r=4$	$r=6$
2	−1.2	−1.1	−1.2	−1.1	−31.4	−31.3	−31.2	−31.3
4	−1.7	−0.9	−0.6	−0.4	−31.0	−30.2	−29.4	−29.0
8	−2.8	−1.7	−0.9	−0.6	−30.6	−29.3	−28.8	−28.3
64	−7.1	−5.4	−3.9	−2.9	−27.1	−25.9	−25.2	−25.1
2^{10}	−11.1	−9.5	−8.2	−7.5	−22.8	−22.0	−21.5	−21.1
2^{29}	−29.9	−28.9	−27.1	−26.5	−3.4	−2.7	−2.1	−1.9
2^{31}	−31.0	−30.2	−28.8	−28.2	−1.2	−0.7	−0.4	−0.2

(c) $\eta^{\min}$ and $\eta^{\max}$ for different M

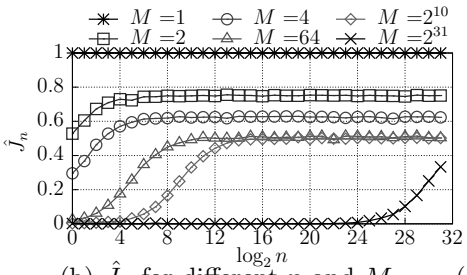
Figure 4.6: Leakage of procedure that checks a guess of secret's residue class modulo M (see Sec. 4.3.1–4.3.2)

monotonically decreases and increase, respectively, as M grows.

4.3.2 Leaking more over multiple rounds

A second way to view the example in Fig. 4.6 is to consider r procedure executions using the same S('secret') (i.e., S_1('secret') = S_2('secret') = ... = S_r('secret')). Our intuition suggests that after $r = M - 1$ executions of the procedure, a smart attacker will have learned everything about S('secret') that it can from *proc*; e.g., by setting C_j('test') = j, the attacker either will have observed some O_j('result') = 1, in which case it knows S('secret') mod $M = j$, or else it knows S('secret') mod $M = 0$. Consistent with that intuition, in Fig. 4.6(c), both $\eta^{\min}$ and $\eta^{\max}$ remain steady for $M = 2$ as r increases, since no new information is available to the attacker after $r = 1$. Similarly, for $M = 4$, $\eta^{\min}$ and $\eta^{\max}$ both increase precipitously (by $\geq 74\%$) from $r = 1$ to $r = 2$ and then begin to flatten out (albeit imperfectly—both are estimated values, after all), which is consistent with this intuition that the attacker should learn no new information past $r = 3$. For $M > 4$, each additional procedure execution provides additional information to the attacker about all secrets and much more about some (namely those for which it learns the residue class mod M).

```
proc (C,I,S)
    if (I('rand') mod M = 0)
        O('result') ← S('secret')
    else
        O('result') ← I('rand')
    return O
```



M	$\log_2 \hat{\eta}^{\min}$	$\log_2 \hat{\eta}^{\max}$
1	0.0	0.0
2	−0.9	−0.5
4	−1.8	−0.8
64	−6.5	−1.1
2^{10}	−10.7	−1.2
2^{31}	−31.8	−1.9

(a) Procedure (b) $\hat{J}_n$ for different n and M (c) $\hat{\eta}^{\min}$ and $\hat{\eta}^{\max}$ for different M

Figure 4.7: An example illustrating leakage dependent on randomness (see Sec. 4.3.3)

Correspondingly, both $\eta^{\min}$ and $\eta^{\max}$ increase monotonically along each of these rows.

4.3.3 Leaking the secret conditioned on randomness

We now illustrate the ability of our technique to measure leakage from a different randomized procedure from that discussed in Fig. 4.2. The procedure, shown in Fig. 4.7(a), returns the secret if a random value is divisible by a constant M and returns that random value otherwise. Clearly, a larger M implies that fewer secret values leak, but those that leak do so completely. This behavior is illustrated by the $\hat{J}_n$ measure shown in Fig. 4.7(b); the leakage is consistently higher for lower values of M. Similarly, while $\hat{\eta}^{\max}$ remains high for all values of M (never dropping below $\frac{1}{4}$), $\hat{\eta}^{\min}$ ranges from $\hat{\eta}^{\min} = 1$ when all secrets are leaked ($M = 1$) to $\hat{\eta}^{\min} \approx 0$ when few secrets are leaked ($M = 2^{31}$).

4.4 Case Studies

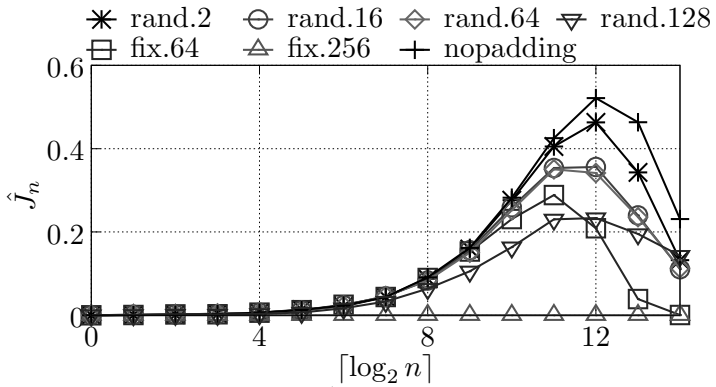
In this section, we illustrate our measurement by applying it to real-world codebases susceptible to the inference of search queries via packet-size observations, inference of secret values due to compression results, and inference of TCP sequence numbers. We claim no novelty in identifying these attacks; all are known and explored in other papers, though not in the particular codebases (or codebase versions) that we examine here and typically only through application-specific analysis. Our contribution lies in showing the applications of our methodology to measuring interference in an application-agnostic way and the impact of alternatives for mitigating that interference.

4.4.1 Traffic analysis on web applications

Packet sizes are a known side channel for reverse engineering search queries and other web content returned from a server, and defenses against this side channel have been studied using various methods of QIF (e.g., [52, 96, 18]). Specifically, a network attacker can often distinguish

Keyword	Trigrams
class	__c _cl cla las ass ss_ s__
code	__c _co cod ode de_ e__
div	__d _di div iv_ v__
the	__t _th the he_ e__
and	__a _an and nd_ d__
title	__t _ti tit itl tle le_ e__

(a) Small database for `Sphinx`



(b) $\hat{J}_n$ for different n

Mitigation	$\log_2 \hat{\eta}^{\min}$	$\log_2 \hat{\eta}^{\max}$
nopadding	−8.3	−1.5
rand.2	−8.4	−1.9
rand.16	−8.5	−2.3
rand.64	−8.5	−2.3
rand.128	−9.0	−2.5
fix.64	−8.5	−3.8
fix.256	nan	nan

"nan" denotes "not a number," i.e., $\hat{\eta}^{\min} = 0$ or $\hat{\eta}^{\max} = 0$

(c) $\hat{\eta}^{\min}$ and $\hat{\eta}^{\max}$ for different mitigations

Figure 4.8: Analysis of auto-complete feature of `Sphinx` and mitigation strategies (see Sec. 4.4.1)

between two queries to a web search engine because the response traffic length is dependent on the query. Even packet padding may not hide all secret information [34].

In this section, we use our methodology to analyze the auto-complete feature of search engines to demonstrate our ability to detect the leakage of the user's query from the network packet sizes. Furthermore, we repeat our analysis after applying mitigations suggested in previous work [34]. This allows us to compare the effectiveness of these mitigations to the original implementation.

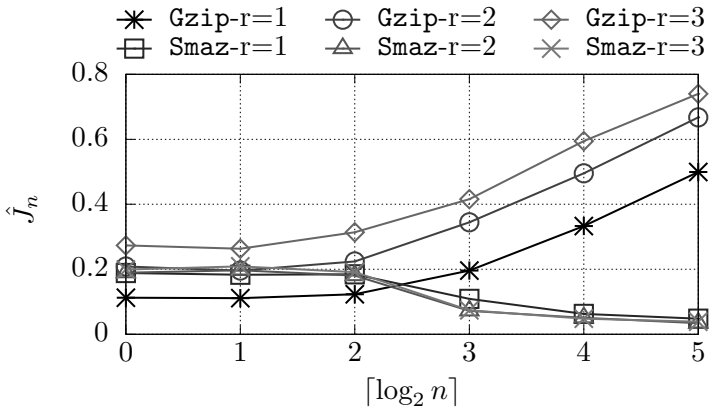
We evaluated a C++ web server called `Sphinx` (`http://sphinxsearch.com/`), which provides PHP APIs for a client to send a query string to the server. The auto-complete feature then returns a list of keywords that best match the query string. To generate the postcondition that characterizes the auto-complete feature, we marked the query string as the secret (i.e., S('secret') is the query string) and the final application response length as the observable (i.e., $\mathit{Vars}_{\mathsf{O}} = \{$'response_length'$\}$), by injecting only two lines into the server's code. In this application, there was no attacker-controlled input and no other input (i.e., $\mathit{Vars}_{\mathsf{C}} = \mathit{Vars}_{\mathsf{I}} = \emptyset$).

Since the auto-complete results depend on the contents of the server database, we simply instantiated the database with an example containing six keywords and 35 query trigrams (see Fig. 4.8(a)).

When provided an input query string of at least three characters, `Sphinx` returns (content containing) the two keywords with the highest "score" based on matching trigrams in the query string to each keyword's associated trigrams. We also limited queries to three characters drawn from $\{\text{'a'}, \ldots, \text{'z'}\}$ ($\{97, \ldots, 122\}$ in ASCII), yielding $26^3 \approx 2^{14}$ possible queries. Note that instantiating the server with a specific database and limiting the query characters and length as described cannot induce our analysis to provide false positives, though it can contribute false negatives.

We experimented with two types of mitigation strategies. *Random padding* is motivated by protocols like SSH that obfuscate traffic lengths by adding a random amount of padding up to some maximum limit to the application response payload. We experimented with padding lengths of up to 2 bytes ('rand.2'), 16 bytes ('rand.16'), 64 bytes ('rand.64'), and 128 bytes ('rand.128'). *Padding to a fixed length* is a second strategy, which increases the length of the application response payload to the nearest multiple of a fixed length. We experimented with padding to a multiple of 64 bytes ('fixed.64') or a multiple of 256 bytes ('fixed.256'). We "implemented" both of these padding strategies by modifying the postcondition $\Pi_{\texttt{Sphinx}}$ to reflect them (vs. modifying the `Sphinx` code directly).

Fig. 4.8(b) shows $\hat{J}_n$ for the random padding strategies and J_n (which is equivalent to $\hat{J}_n$ since $Vars_{\mathsf{I}} = \emptyset$) for the original, 'fixed.64', and 'fixed.256' strategies. Here, 'nopadding' is the result for original auto-complete in `Sphinx`. In addition, Fig. 4.8(c) shows the measure $\hat{\eta}^{\min}$ and $\hat{\eta}^{\max}$ for each strategy. Only 'fixed.256' reaches zero leakage, indicated by 'nan' ('not a number'), since any result from `Sphinx` populated with the database in Fig. 4.8(a) fit within 256 bytes and so resulted in a padded payload of that length. Comparing different padding mechanisms, our measures $\hat{\eta}^{\min}$ and $\hat{\eta}^{\max}$ show results consistent with the intuitive order of the different mitigation strategies in terms of their effectiveness in preventing leakage. Our results suggest that 'nopadding' leaks the most, followed by 'rand.2.' The configuration 'rand.16' was only very slightly worse than 'rand.64', and 'fix.64', which provided similar protection for this setup, and 'rand.128' provided better protection than all others except 'fixed.256.' These results demonstrate the power of our methodology for enabling comparisons of the benefits of different amounts of padding *for this database*. For example, our analysis shows that for this database, 'rand.64' provides little security benefit compared to 'rand.16', despite adding 3× more padding in expectation.



(a) J_n for different n and r

Procedure	$\log_2 \hat{\eta}^{\min}$			$\log_2 \hat{\eta}^{\max}$		
	$r=1$	$r=2$	$r=3$	$r=1$	$r=2$	$r=3$
`Gzip`	-2.04	-1.22	-0.85	-1.00	-0.58	-0.43
`Smaz`	-1.58	-1.55	-1.54	-3.73	-4.02	-3.95

(b) $\hat{\eta}^{\max}$ and $\hat{\eta}^{\max}$ for different r

Figure 4.9: Leakage from `Gzip` and `Smaz` (see Sec. 4.4.2)

4.4.2 Leakage in compression algorithms

Our methodology is powerful in accounting for attacker-controlled inputs, and in this section we demonstrate the benefits of this capability by applying it to detect `CRIME` attacks [53, 4]. A `CRIME` vulnerability arises when a web client applies "unsafe" compression prior to transmitting a request over TLS. HTTP requests can carry information (e.g., the URL parameters) that an attacker can induce; e.g., if the client visits an attacker-controlled website, then the attacker can induce requests from the client to another, target website with URL parameters that the attacker sets. By observing the lengths of compressed requests to the target website, the attacker can deduce whether the attacker-controlled input shares a substring with a secret contained in the request (e.g., the client's cookie for the target website) that the attacker is unable to observe directly. To be concrete, if the attacker-induced request to the target website is `http://target.com?username=`*name* then the request will compress better if *name* is a prefix of the client's cookie for target.com.

`CRIME` attacks utilize the property of an adaptive compression algorithm that the encoding dictionary is dependent on both the secret and attacker-controlled variables. As suggested by Alawatugoda et al. [4], a possible mitigation is to separate the compression for the secret and the other parts of the plaintext or to use a fixed-dictionary compression algorithm such as `Smaz` [85]. The latter mitigation, though an improvement, removes the influence of the attacker-controlled

input only on the compression dictionary. Consider a two-byte plaintext ab whose first character is secret. If a is '`a`', then this two-byte word will be compressed if b is '`t`' and will be left unchanged if b is '`y`', assuming '`at`' is in the dictionary but '`ay`' is not. Thus, the leakage should not be zero even if a fixed-dictionary algorithm is used.

To analyze this scenario in our framework, we modeled the input for `Gzip` and `Smaz` to be of the form

'`http://target.com/?secret=`' + S('secret') + I('suffix') + '`,username=secret=`' + C('input')

where '+' denotes concatenation. Here, S('secret') and C('input') were each one byte, I('suffix') was two bytes, and the attacker-observable variable was the length of the compressed string. Each byte was allowed to range over 'a', ..., 'z' and '0',...,'9'. The S('secret') byte after the first '`secret=`' plays an analogous role to the client cookie in a `CRIME` attack, i.e., as the secret to be guessed by the attacker, and the '`secret=`' immediately following '`username=`' serves as a prefix to match the first instance of '`secret=`.'

We applied our tool to analyze the leakage susceptibility of `Gzip`-1.2.4 and `Smaz` in this configuration, executed up to three times ($r \in \{1, 2, 3\}$) with the same secret. Our results are shown in Fig. 4.9. Our results show that for one execution ($r = 1$), `Smaz` is no better than `Gzip`. That is, $\eta^{\max}$ and $\eta^{\max}$ in Fig. 4.9(b) suggests that `Smaz` leaks less information about some secrets but some information about more secret values versus `Gzip`; as mentioned above, `Smaz` can leak information about a secret value if it composes a word in its dictionary, as well. However, the strength of `Smaz` is revealed as r grows, since its leakage remains unchanged. In contrast, the leakage of `Gzip` grows with r, essentially matching that of `Smaz` at $r = 2$ and surpassing it at $r = 3$ (in terms of $\eta^{\min}$). This occurs because in each execution of `Gzip`, the attacker has the latitude to select a different value for C('input') and then observe that selection's impact on the length of the compressed string (which in general will change). In contrast, the leakage of `Smaz` is independent of the adversary's choice for C('input'), and so additional executions do not leak any additional information.

As discussed at the end of Sec. 4.2.2, a side effect of our methodology is identifying some example $\langle \mathsf{C}, \mathsf{O} \rangle$ pairs that lie in $Y_S \setminus Y_{S'}$ or $Y_{S'} \setminus Y_S$ for samples S, S' of secrets, which can help in diagnosing a leak. For example in Table 4.2, for `Gzip` in the $r = 1$ case, our tool identified the $\langle \mathsf{C}, \mathsf{O} \rangle$ pair with C('input') = 'c' and O('length') = 66 as being in $Y_S \setminus Y_{S'}$ for a sampled S,

S' where $S \ni$ 'c' = S('secret') and I('suffix') = 'oo'.[8] As such, the developer now knows that this $\langle \mathsf{C}, \mathsf{O} \rangle$ pair is consistent with no secret in S'. Similarly, for `Smaz` our tool identified the pair $\langle \mathsf{C}, \mathsf{O} \rangle$ with C('input') = 'r' and O('length') = 36 as being in $Y_S \setminus Y_{S'}$ for a sampled S, S' where $S \ni$ 'f' = S('secret') and I('suffix') = 'or'.

Table 4.2: Examples from $Y_S \setminus Y_{S'}$ for samples S, S' ($r = 1$) in `CRIME` attacks

	C('input')	O('length')	S ('secret')	I('suffix')
`Gzip`	'c'	66	'c'	'oo'
`Smaz`	'r'	36	'f'	'or'

4.4.3 Linux TCP sequence number leakage

Known side channels in some TCP implementations leak TCP sequence and acknowledgment numbers [72, 79]. In some cases, these side channels can be used by off-path attackers to terminate or inject malicious payload into connections [13, 79]. The origin of these attacks is shared network counters (e.g., `linux_mib` and `tcp_mib`) that are used to record connection statistics across different connections in the same network namespace.

These counters have been implicated in numerous side channels since version 2.0 of the Linux kernel [64]. For example, the code snippet (without the patch in Lines 6–10) in Fig. 4.10 leaks the secret `tp->rcv_nxt` in Linux-3.18 TCP. Here, the attacker controls the `skb` input and so the value `TCP_SKB_CB(skb)->seq` that is compared to `tp->rcv_nxt` on Line 5. Based on this comparison, the `NET_INC_STATS_BH` procedure increments an attacker-observable counter indicated by `LINUX_MIB_DELAYEDACKLOST` (Line 11). If the attacker can repeatedly cause the procedure in Fig. 4.10 to be invoked with inputs `skb` of its choice, it can use binary search to infer `tp->rcv_nxt` within 32 executions [79].

The most straightforward mitigation for this leakage is to disable the public counters. This will stop the leakage, but will disable some mechanisms such as audit logging. Another potential mitigation is to increase the difficulty of increasing the public counter, by adding additional checking related to more secret variables. For example, before increasing the `LINUX_MIB_DELAYEDACKLOST`

[8] The output length of 66 exceeds the length of the input string because `Gzip` adds a header to the output. `Smaz` attaches no such header.

```
 1 void tcp_send_dupack(struct sock *sk,
 2                       const struct sk_buff *skb) {
 3  struct tcp_sock *tp = tcp_sk(sk);
 4  if (TCP_SKB_CB(skb)->end_seq != TCP_SKB_CB(skb)->seq &&
 5     before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt)) {
 6 +  if (before(TCP_SKB_CB(skb)->ack_seq, tp->snd_una - tp->max_window)
 7 +       || after(TCP_SKB_CB(skb)->ack_seq, tp->snd_nxt)) {
 8 +     tcp_send_ack(sk);
 9 +     return;
10 +  }
11     NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_DELAYEDACKLOST);
12     ...
13  }
14  tcp_send_ack(sk);
15 }
```

Figure 4.10: A code snippet vulnerable to leaking the TCP sequence number in linux 3.18; lines marked '+' indicate a hypothetical patch with which we experimented (see Sec. 4.4.3)

counter, the procedure could also check for correct acknowledgment numbers, as shown in the patch in Lines 6–10. As far as we know, our study is the first to compare these potential mitigations for TCP sequence and acknowledgment number leakage.

To analyze the information leakage in this example, we compiled a user-mode Linux kernel [30] as a library. Our target procedure for analysis was `tcp_rcv_established`, which is of the form

```
void tcp_rcv_established(struct sock *sk,
  struct sk_buff *skb,
  const struct tcphdr *th,
  unsigned int len) {
  struct tcp_sock* tp = (struct tcp_sock*) sk;
  ...
}
```

The inputs for `tcp_rcv_established` have many constraints among them when passed in, for instance

$$\texttt{TCP\_SKB\_CB(skb)->seq} < \texttt{TCP\_SKB\_CB(skb)->end\_seq}$$

$$\texttt{tp->rcv\_wnd} \leq \texttt{MAX\_TCP\_WINDOW}$$

$$\texttt{tp->snd\_wnd} \leq \texttt{MAX\_TCP\_WINDOW}$$

To generate constraints for the inputs to `tcp_rcv_established`, we applied symbolic execution to the procedures `fill_packet` and `tcp_init_sock`. Symbolic buffers to represent these inputs and their associated constraints were then assembled within `tcp_rcv_established`. We also stubbed

out several procedure calls[9] within `tcp_rcv_established`, causing each to simply return a symbolic buffer so as to avoid symbolically executing it, since doing so introduced problems for `KLEE` (e.g., dereferencing symbolic pointers).

After generating the postcondition for the procedure `tcp_rcv_established`, we defined the attacker-controlled inputs to be

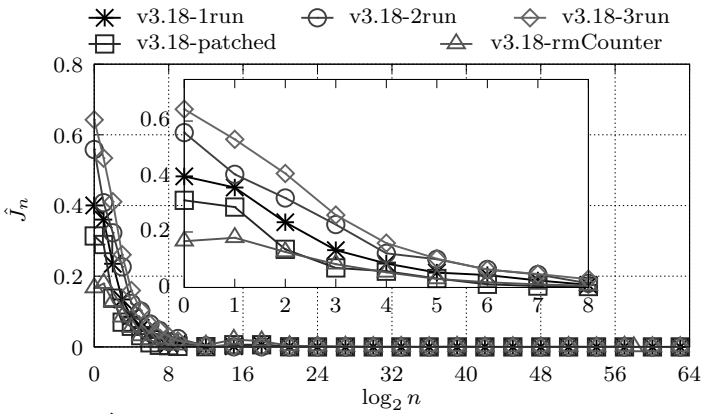
$$\begin{aligned} \mathit{Vars}_{\mathsf{C}} = \{&\texttt{TCP\_SKB\_CB(skb)->seq},\\ &\texttt{TCP\_SKB\_CB(skb)->end\_seq},\\ &\texttt{TCP\_SKB\_CB(skb)->ack\_seq},\\ &\texttt{tcp\_flag\_word(th)}\} \end{aligned}$$

(each four bytes) and the attacker-observable variables to be $\mathit{Vars}_{\mathsf{O}} = \{\texttt{linux\_mib}, \texttt{tcp\_mib}\}$. All fields of constrained input structures (e.g., `tp->snd_una` and `tp->max_window`) not covered by $\mathit{Vars}_{\mathsf{C}}$ and $\mathit{Vars}_{\mathsf{O}}$ were added to $\mathit{Vars}_{\mathsf{I}}$, with the secret variables[10] being `tp->rcv_nxt` and `tp->snd_nxt` (each four bytes). We conducted single-execution ($r = 1$, denoted 'v3.18-1run'), two-execution ($r = 2$, denoted 'v3.18-2run') and three-execution ($r = 3$, denoted 'v3.18-3run') leakage analysis. In the multi-execution analysis, we assumed `*sk` to be the same in multiple executions (I_1('`*sk`') $= \mathsf{I}_2$('`*sk`') $= \ldots = \mathsf{I}_r$('`*sk`')) since its fields used in `tcp_rcv_established` would be unchanged or, if changed, would be changed predictably.

The results from this analysis are shown in Fig. 4.11. The inset graph in Fig. 4.11(a) is a magnification of the portion of the curve in the interval $[0, 8]$ on the horizontal axis. Specifically, the highest leakage resulted from 'v3.18-3run', followed by 'v3.18-2run' and 'v3.18-1run', as indicated by the $\hat{J}_n$ curves in Fig. 4.11(a) and the $\hat{\eta}^{\min}$ and $\hat{\eta}^{\max}$ measures in Fig. 4.11(b). This shows the potential for the attacker to extract more information about the secrets `tp->rcv_nxt` and `tp->snd_nxt` using multiple executions. This is consistent with the observation that a smart attacker could utilize this side channel to infer one bit per execution [79].

To alleviate this leak, we applied a hypothetical patch shown in Fig. 4.10 that checks another

[9]Specifically, we stubbed out `get_seconds`, `current_thread_info`, `tcp_options_write`, `tcp_sendmsg`, `prandom_bytes`, `current_thread_info`, `tcp_parse_options`, and `tcp_checksum_complete_user`.
[10]Though we have described our framework so far using one secret variable, it extends trivially to more.



(a) $\hat{J}_n$ per n and version of `tcp_rcv_established`

Version	$\log_2 \hat{\eta}^{\min}$	$\log_2 \hat{\eta}^{\max}$
v3.18-1run	−1.6	−63.0
v3.18-patched	−2.1	−64.1
v3.18-rmCounter	−4.0	−65.6
v3.18-2run	−1.0	−62.1
v3.18-3run	−0.7	−61.6

(b) $\hat{\eta}^{\min}$ and $\hat{\eta}^{\max}$ for versions of `tcp_rcv_established`

Figure 4.11: TCP sequence-number leakage (see Sec. 4.4.3)

secret value `tp->snd_nxt` before incrementing the counter for `LINUX_MIB_DELAYEDACKLOST`. Our analysis results (for $r = 1$ execution, denoted 'v3.18-patched') in Fig. 4.11 shows that the patch alleviated the leakage somewhat. We also tried just deleting Line 5-11 from the original (unpatched) code in Fig. 4.10. As shown in Fig. 4.11, this version (denoted 'v3.18-rmCounter') evidently has lower leakage than 'v3.18-patched'. In considering these mitigations, we stress that our patch addressed only the leakage arising from Line 5, and not all sources that leak information about `tp->rcv_nxt` or `tp->snd_nxt` (which are numerous, see Chen et al. [17]). Our results suggest, however, that our methodology could guide developers in mitigating leaks in their code.

4.4.4 Performance

Performance of our tool involves two major components, namely the time to compute the postcondition Π_{proc} via symbolic execution, and the time to calculate J_n or $\hat{J}_n$ for different n starting from Π_{proc}. Postcondition generation is not a topic in which we innovate, and so we defer discussion of its costs in our case studies to Sec. 4.2.1. Here we focus on the costs of calculating J_n or $\hat{J}_n$ for different n starting from Π_{proc}.

Starting from Π_{proc}, the computation of J_n or $\hat{J}_n$ can be parallelized almost arbitrarily. Not only can J_n or $\hat{J}_n$ for each n be computed independently, but even for a single value of n, the estimation of $J(S, S')$ or $\hat{J}(S, S')$ can be computed for each pair of sampled sets S, S' and each estimation iteration independently. In Fig. 4.12, we report the *average* estimation time per sample pair, which indicates that all case studies could finish one estimation in (4.13) for one sample pair within about one minute. As such, the speed of calculating final pair $\eta^{\min}$ and $\eta^{\max}$ is limited

Sec.	Procedure	$J(S,S')$	$\hat{J}(S,S')$	J_n	$\hat{J}_n$
4.4.1	Auto-complete (nopadding)	34ms	56ms	5m	7m
4.4.1	Auto-complete (fix.64)	48ms	65ms	6m	8m
4.4.1	Auto-complete (fix.256)	43ms	57ms	6m	7m
4.4.1	Auto-complete (rand.64)		1.2s		15m
4.4.2	`Gzip`		26s		4h
4.4.2	`Smaz`		40s		10h
4.4.3	v3.18-1run		73s		20h
4.4.3	v3.18-patched		67s		20h
4.4.3	v3.18-rmCounter		50s		19h

Figure 4.12: Average time per estimate ($J(S,S')$ or $\hat{J}(S,S')$) and most expensive overall time (J_n or $\hat{J}_n$) for case studies

primarily by the number of processors available for the computation.

In our experiments, performed on a DELL PowerEdge R815 server with 2.3GHz AMD Opteron 6376 processors and 128GB memory, we computed J_n or $\hat{J}_n$ per value of n on its own core. As reported in the last two columns of Fig. 4.12, the time to do so for the *most expensive* value of n ranged from roughly 15m for the auto-complete procedure of Sec. 4.4.1 to about 20h for the Linux TCP implementations of Sec. 4.4.3. For several of our case studies (see Fig. 4.12), we experimented with calculating $\hat{J}_n$ even when J_n was sufficient, and found its estimation to cost $\leq 2\times$ that of estimating J_n, due to the duplication of Π_{proc} in $\hat{X}_p$.

To place the above numbers in some context, the $\approx$ 20h (for the worst n, without parallelization) dedicated to computing a value of J_n in the Linux TCP case study of Sec. 4.4.3 involved a procedure *proc* of which 165 bytes of its inputs were somehow used in the procedure. A naive alternative to our design in which all possible inputs are enumerated and run through the procedure to compute its outputs (and interference measured from these input-output pairs, perhaps as we do) would therefore involve enumerating 2^{1320} possible inputs, which is obviously impractical.

In this light, our technique that performs interference analysis for real codebases in the time-frame of minutes-to-hours (and far faster with parallelization) is a dramatic improvement. Moreover, these results are likely to only improve with advances in symbolic execution and model counting. Even our experimentation with various optimizations for postcondition generation and model counting was not exhaustive. That said, the results above suggest that the costs of our approach are likely to remain sufficiently high for real codebases to preclude its use for interactive analysis by human programmers. Rather, we expect that our analysis could be run as a diagnostic technique

overnight, for example.

4.5 Discussion and Limitations

The static method to quantify noninterference builds from two tasks that are recognized, difficult challenges in computer science. The first is the construction of a logical postcondition Π_{proc} for a procedure *proc*, for which we leverage symbolic execution. As such, our technique inherits the limitations of existing symbolic execution tools and those incumbent on generating postconditions, more generally. For example, symbolic execution is difficult to scale to some procedures, and challenges involving symbolic pointers and unbounded loops can require workarounds, as they did in our TCP case study (Sec. 4.4.3). The second challenge problem underpinning our methodology is model counting, which is #P-complete. We are optimistic that future improvements in these areas will be amenable to adoption within our methodology. While the resulting tool is not yet quick enough to support interactive use, it is positioned to benefit from advances in symbolic execution and approximate model counting, both active areas of research.

Our approach is powerful in that it can be applied to scenarios in which the distributions of inputs—whether they be attacker controlled or other—are unknown, and this is often the case in practice. In some cases, the input distributions are unknowable, especially for $Vars_{\mathsf{C}}$. In others, they may be knowable but require considerable empirical data to estimate (e.g., the distributions of user-input search terms, in a context like that of Sec. 4.4.1). That said, because it is insensitive to these distributions, it does not offer an immediate way to accommodate these distributions if they are known. Still, our methodology allows these inputs to be accounted for in a principled way, in contrast to others that either disallow them or assign them heuristically.

Our measure does not take into account computational limits on the attacker, as is typically assumed by cryptographic algorithms. As such, our measure is best viewed as a measure of information-theoretic security, where the attacker has unlimited computation power. For example, assuming it were possible to generate our measure for a digital signature algorithm, our measure would typically indicate that a signature and public key divulges the private key, even if the private key is intractable to compute. That said, due to hardness in solving a cryptographic problem with a solver, secrets hidden by cryptography would typically be intractable to generate postconditions for.

4.6 Summary

In this chapter we have suggested a static method for assessing interference and attempts to mitigate it. Informally, noninterference is achieved when the output produced by a procedure in response to an adversary's input is unaffected by secret values that the adversary is not authorized to observe. Following this intuition, we have developed an automatic tool to estimate the number of pairs of attacker-controlled inputs and attacker-observable outputs that are possible, conditioned on the secret being limited to a particular sample. The discovery of such pairs that are possible for one sample but not another reveals interference.

We clarified the effectiveness of our strategy both on artificial examples (Sec. 4.3) and on real-world codebases (Sec. 4.4). Specifically, we evaluated leakage in the `Sphinx` auto-complete feature of its search interface due to its response sizes, and the effectiveness of a variety of mitigations (Sec. 4.4.1); the `CRIME` vulnerabilities of adaptive compression in `Gzip` and fixed-dictionary compression in `Smaz` (Sec. 4.4.2); and leakage of TCP sequence numbers in Linux and the effectiveness of two mitigations of our own design (Sec. 4.4.3). Within these contexts we also explored leakage over a single procedure execution and over many, and showed that our framework allowed for a useful comparison of how procedures leaked data as the number of executions grows.

Central to our methodology's ability to scale to real codebases is our expression of leakage assessment within a framework that permits the use of approximate model counting (and specifically hash-based model counting).

CHAPTER 5: DECLASSIFICATION AND INTERPRETABILITY

Based on the quantitative noninterference measure, this chapter permits information to be declassified to focus on actual unintended leakage and interprets leakage measurements for the analyst in terms of simple rules that characterize when leakage occurs.

While noninterference measurement for arbitrary computations remains out of reach, in this chapter we address the shortcomings in our measuring framework within a particularly important and complex domain, namely information leaks arising in hardware processors. Leakage of software secrets due to processor optimization have attracted massive attention in recent years, especially since the discovery of vulnerabilities arising due to the footprint of speculative executions in processor caches (SPECTRE [57], MELTDOWN [65], and variants). Even though many defenses (e.g., [91, 101, 93, 90]) have been proposed to restrict cache-based side channels, we are aware of no measurement methodology to compare designs and evaluate their effectiveness, working directly from their Verilog specifications. Adapting the framework proposed in Chapter 4 to do so, however, appears difficult, due to the sheer complexity of modern processor designs.

In this chapter, we present a methodology that does so, using three key methodological advances:

- Since generating a logical postcondition for a processor's execution of a program en masse is intractable, we devise a method to build the postcondition one cycle at a time. To build single-cycle formulas, we abandon symbolic execution, as we found that applying it to hardware designs induces significant path explosion for even one CPU cycle. Instead, we extract the single-cycle formulas without solving for feasible paths, and then leverage a number of aggressive optimizations when stitching single-cycle formulas together to build the postcondition for the processor's multi-cycle execution. In doing so, we need to be careful that these optimizations preserve the number of assignments to relevant variables across all solutions, a socalled *projected model counting* problem.
- Since some leakage is inevitable, our methodology enables analysts to *declassify* certain information, thereby focusing the measurement on any *other* leakage that might be occurring, i.e.,

leakage that cannot be inferred from the declassified information. For systems as complex as modern processors, this ability is essential to permit analysts to decompose and analyze leakage in a piecemeal fashion.

- The sheer complexity of processor designs means that once leakage is measured, the exact conditions that cause this leakage might not immediately be evident. Our methodology therefore incorporates a method of *interpreting* the leakage, i.e., providing simple rules that indicate circumstances in which leakage will (or will not) occur. Each such rule is additionally accompanied by a precision and recall, so that analysts can prioritize the rules they address.

Due to the focus of our methodology on support for declassification and interpretability, we call our tool that realizes it DINoMe (for "Declassification and Interpretability for Noninterference Measurement").

To evaluate DINoMe, we apply it to evaluate leakage arising from execution on a RISC-V BOOM core [14], a state-of-the-art public domain processor design. Our improvements to generating logical postconditions for execution permit DINoMe to do so for more than 100 cycles of this core. This, in turn, permits us to evaluate leakage from cache-based side channels (Prime+Probe [73] and Flush+Reload [95]) in a variety of scenarios, including cryptographic key leakage in sliding-window based modular exponentiation (e.g., [75, 3]), leakage of secrets due to speculative execution [57, 65], and how this leakage is (incompletely) mitigated by proposed improvements such as ScatterCache [93] and PhantomCache [90]. In each case, we measure interference and additionally generate rules to explain why the leakage occurs, and in some cases refine our view of the leakage using declassification. Our performance evaluation of DINoMe indicates that while it is not fast enough to support interactive use by the analyst, these types of analyses complete in times ranging from seconds to under 15 minutes (using horizontal scaling), after an initial phase to assemble the logical postcondition of up to (only) two hours on (only) a single core.

The rest of this chapter is structured as follows. We introduce our declassification in Sec. 5.1. We then present our method for interpreting leakage in Sec. 5.2. We address various implementation challenges in Sec. 5.3, and then evaluate our tool, DINoMe, through several case studies in Sec. 5.4. We discuss limitations in Sec. 5.5 and the summarize this chapter's results in Sec. 5.6.

5.1 Measuring Interference with Declassification

Some sources of information leakage may be desirable or inevitable; e.g., the results of a password check will tell whether an input is the correct password, and so will "leak" information about the correct password. To exclude intended leakage from the analysis, it will be helpful to provide a method to exempt some identified information leakages specified by the analyst, allowing the analysis to focus on the leakage that remains. Specifically, our methodology seeks to assess the degree to which a procedure permits secrets to be distinguished by the attacker using attacker-observable and declassified information but not by the declassified information alone.

Let $\Delta \leftarrow \delta(\mathsf{C}, \mathsf{I}, \mathsf{S})$ denote the allowed information exposure (e.g., for a password checker, Δ is whether the input is the legitimate password), and let

$$\Pi_{proc,\delta}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{O}, \Delta) \leftarrow \Pi_{proc}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{O}) \wedge \Pi_{\delta}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \Delta)$$

where $\Pi_{\delta}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \Delta)$ is a logical postcondition for δ that relates Δ to C, I, and S. Then, we can define the attacker's accessible set Y_S^δ of $\langle \mathsf{C}, \mathsf{O}, \Delta \rangle$ tuples and allowed accessible set D_S^δ consistent with chosen secret set S by

$$\begin{aligned}
X_S^\delta &= \left\{ \langle \mathsf{C}, \mathsf{O}, \Delta, \mathsf{I} \rangle \,\middle|\, \exists \mathsf{S} : \mathsf{S}(svar) \in S \wedge \ \Pi_{proc,\delta}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{O}, \Delta) \right\} \\
Y_S^\delta &= \left\{ \langle \mathsf{C}, \mathsf{O}, \Delta \rangle \,\middle|\, \exists \mathsf{I} : \langle \mathsf{C}, \mathsf{O}, \Delta, \mathsf{I} \rangle \in X_S^\delta \right\} \\
D_S^\delta &= \left\{ \langle \mathsf{C}, \Delta \rangle \,\middle|\, \exists \mathsf{O}, \mathsf{I} : \langle \mathsf{C}, \mathsf{O}, \Delta, \mathsf{I} \rangle \in X_S^\delta \right\}
\end{aligned}$$

Since the declassified information is allowed to leak, we are concerned only with cases where the secret is distinguishable by $\langle \mathsf{C}, \mathsf{O}, \Delta \rangle$ but not by $\langle \mathsf{C}, \Delta \rangle$. Here, we define a set $\tilde{X}_{S,S'}^\delta$ to include the tuples $\langle \mathsf{C}, \mathsf{O}, \Delta \rangle$ that leak whether the secret is in S or S', assuming $\langle \mathsf{C}, \Delta \rangle$ is equivalent.

$$\check{X}_{S,S'}^\delta = \left\{ \langle \mathsf{C}, \mathsf{O}, \Delta, \mathsf{I} \rangle \,\middle|\, \langle \mathsf{C}, \mathsf{O}, \Delta, \mathsf{I} \rangle \in X_S^\delta \cup X_{S'}^\delta \wedge \langle \mathsf{C}, \Delta \rangle \in D_S^\delta \cap D_{S'}^\delta \right\} \tag{5.1}$$

$$\hat{X}_{S,S'}^\delta = \left\{ \langle \mathsf{C}, \mathsf{O}, \Delta, \mathsf{I} \rangle \,\middle|\, \langle \mathsf{C}, \mathsf{O}, \Delta, \mathsf{I} \rangle \in \check{X}_{S,S'}^\delta \wedge \ \langle \mathsf{C}, \mathsf{O}, \Delta \rangle \in Y_S^\delta \cap Y_{S'}^\delta \right\} \tag{5.2}$$

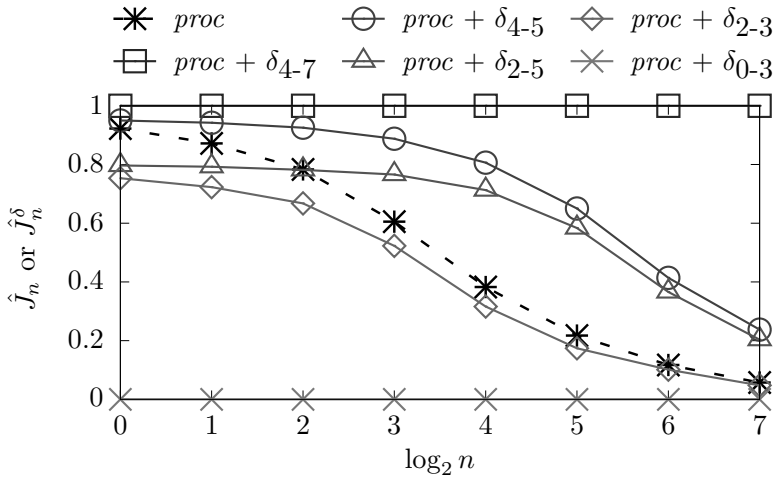
$$\tilde{X}_{S,S'}^\delta = \check{X}_{S,S'}^\delta \setminus \hat{X}_{S,S'}^\delta \tag{5.3}$$

$proc(\mathsf{C}, \mathsf{I}, \mathsf{S})$
  $\mathsf{O}(ovar) \leftarrow \mathsf{S}(svar)[0:3]$

(a) An artificial procedure

$\delta_{i\text{-}j}(\mathsf{C}, \mathsf{I}, \mathsf{S})$
  $\Delta(dvar) \leftarrow \mathsf{S}(svar)[i:j]$

(b) Declassification policy



(c) Measurement with vs. without declassification

Figure 5.1: Declassification example

Thus, we can use an alternative metric

$$\hat{J}^{\delta}(S, S') = \frac{\left|\tilde{X}^{\delta}_{S,S'}\right|}{\left|\check{X}^{\delta}_{S,S'}\right|} \tag{5.4}$$

$$\hat{J}^{\delta}_{n} = \underset{\substack{S, S' : |S| = |S'| = n \\ \wedge\ S \cap S' = \emptyset}}{\text{avg}} \hat{J}^{\delta}(S, S') \tag{5.5}$$

To illustrate the use of $\hat{J}^{\delta}_{n}$, consider the simple procedure shown in Fig. 5.1(a). In this procedure, $\mathsf{S}(svar)$ is an 8-bit value, and *proc* simply outputs the lowest 4 bits as $\mathsf{O}(ovar)$. The declassification policy shown in Fig. 5.1(b) allows the i-th to j-th bits of $\mathsf{S}(svar)$ to be released. We evaluate $\hat{J}^{\delta}_{n}$ with differently parameterized declassification policies in Fig. 5.1(c). Specifically, when the lowest 4 bits ($i = 0$, $j = 3$) are declassified, then the additional leakage from *proc* is nothing, which is demonstrated by the "$proc + \delta_{0\text{-}3}$" curve. When the declassification policy declassifies all but the lowest 4 bits ($i = 4$, $j = 7$), then the additional leakage by *proc* is maximized, as shown by the "$proc + \delta_{4\text{-}7}$" curve. Intuitively, if $\mathsf{O}(ovar)$ and Δ('info') do not overlap (e.g., "$proc + \delta_{4\text{-}7}$" and "$proc + \delta_{4\text{-}5}$"), then the $\hat{J}^{\delta}_{n}$ curve should be higher than $\hat{J}_n$, whereas if $\mathsf{O}(ovar)$ includes all of Δ('info') (e.g., "$proc + \delta_{0\text{-}3}$" and "$proc + \delta_{0\text{-}1}$"), then $\hat{J}^{\delta}_{n}$ should be lower than $\hat{J}_n$. A hybrid case

occurs when $\mathsf{O}(ovar)$ includes a portion of Δ('info') (e.g., "$proc + \delta_{2\text{-}5}$"), where $\hat{J}_n^\delta$ is lower than $\hat{J}_n$ when n is small but becomes larger when n is large. This is consistent with the interpretation that $\hat{J}_n^\delta$ with small n primarily reflects the number of secret values for which interference occurs [100]; e.g., when $n = 1$, two secret values share bits 0–1 (and so cannot be distinguished by bits 0–3 after declassifying bits 2–5) in 25% of cases, but share bits 0–3 (and so cannot be distinguished using them) in only 6.25% of cases. Larger n, in contrast, better reflects the amount of leakage that occurs [100]. For example, in a random partition of all 2^8 values into sets S and S' of equal size (i.e., $n = 2^7$), every value for bits 2–5 is represented in both S and S' with high probability. In conjunction with the additional bits 0–1 output in O (yielding six bits of the secret value in total), however, these bits give the attacker greater distinguishing power than do bits 0–3 alone.

5.2 Interpreting Leakage

Our defined quantitative leakage metric can measure the interference of a secret with the outputs observable to the attacker. For this to be useful to an analyst, however, it is helpful to provide guidance as to *why* this leakage occurs. Specifically, while the conditions under which leakage occurs are already represented in the postcondition, it may be difficult to understand the formula without further help. In this section, we provide a method for generating short and understandable rules to explain the leakage to a user.

5.2.1 Noninterference and interference tuples

Our first step toward providing an intuitive explanation for the leakage that occurs is to train a binary classifier to classify 4-tuples $\langle \mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{S}' \rangle$ into those that illustrate leakage occurring (i.e., that permit the attacker to distinguish $\mathsf{S}(svar)$ and $\mathsf{S}'(svar)$ from the resulting output O) and those that do not. When using declassification, the interference tuples should only include those where the secrets can be distinguished using $\mathsf{C}, \mathsf{O}, \Delta$ but not using just C, Δ.

More specifically, we define the interference set IS based on (5.3). That is, when the attacker chooses C, if an observable value is feasible for $\langle \mathsf{I}, \mathsf{S} \rangle$ for some I but is never possible for $\langle \mathsf{I}', \mathsf{S}' \rangle$ for any I' that shares a declassification value with $\langle \mathsf{I}, \mathsf{S} \rangle$, then $\langle \mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{S}' \rangle$ is added to IS:

$$IS = \left\{ \langle \mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{S}' \rangle \;\middle|\; \begin{array}{rl} \exists \mathsf{O}, \Delta & : \quad \langle \mathsf{C}, \mathsf{O}, \Delta, \mathsf{I} \rangle \in X_S^\delta \\ & \wedge \; \langle \mathsf{C}, \Delta \rangle \in D_{S'}^\delta \cap D_{S'}^\delta \\ & \wedge \langle \mathsf{C}, \mathsf{O}, \Delta \rangle \in Y_S^\delta \setminus Y_{S'}^\delta \end{array} \right\} \tag{5.6}$$

where $S = \{\mathsf{S}(svar)\}$ and $S' = \{\mathsf{S}'(svar)\}$.

The noninterference set NS should include two types of tuples. For an attacker-chosen C, if there is a observable value O that is feasible for an $\langle \mathsf{I}, \mathsf{S} \rangle$ pair and an $\langle \mathsf{I}', \mathsf{S}' \rangle$ pair, tuple $\langle \mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{S}' \rangle$ belongs to NS as it is an example where no interference occurs. In addition, for an attacker-chosen C, if there is a declassification value Δ that is feasible for $\langle \mathsf{I}, \mathsf{S} \rangle$ but not $\langle \mathsf{I}', \mathsf{S}' \rangle$ for any I', then $\langle \mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{S}' \rangle$ should also be added to NS, as S and S' can already be distinguished using the declassified value:

$$\begin{aligned} NS = & \left\{ \langle \mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{S}' \rangle \;\middle|\; \exists \mathsf{O}, \Delta : \; \langle \mathsf{C}, \mathsf{O}, \Delta, \mathsf{I} \rangle \in X_S^\delta \wedge \; \langle \mathsf{C}, \mathsf{O}, \Delta \rangle \in Y_S^\delta \cap Y_{S'}^\delta \right\} \\ & \cup \left\{ \langle \mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{S}' \rangle \;\middle|\; \exists \mathsf{O}, \Delta : \; \langle \mathsf{C}, \mathsf{O}, \Delta, \mathsf{I} \rangle \in X_S^\delta \wedge \; \langle \mathsf{C}, \Delta \rangle \in D_S^\delta \setminus D_{S'}^\delta \right\} \end{aligned} \tag{5.7}$$

where $S = \{\mathsf{S}(svar)\}$ and $S' = \{\mathsf{S}'(svar)\}$.

Since NS and IS are large in practical scenarios, enumerating all tuples is generally unfeasible. Instead, we generate samples in each set to train a machine learning model, from which explanations of the leakage will be extracted (as described below). Doing so with modern `SAT` solvers, however, typically results in samples that cover NS and IS unevenly, since solvers generally enumerate the next solution by simply adding a conflict constraint to block out previous solutions; as a result, the next solution found is typically close to the previous. Another drawback of using this "blocking" method to sample is that we cannot parallelize the sampling.

For this reason, we sample from NS and IS using hash-based sampling (cf., Chapter 4). Specifically, we sample a limited number of solutions by adding a random universal hashing constraint to the formula given to the solver. Due to the hash function's universality, we can run multiple samplers in parallel to generate a large number of uniformly distributed solutions. In most cases, the sizes of the sampled sets $\hat{NS}$ and $\hat{IS}$ differ either due to differences in the sizes of NS and IS or due to the solving difficulty of one set compared to the other. We associate a sample weight to each element to ensure that the weight of each set is equal in the training process described below.

5.2.2 Interpretation through a rule-based method

Given $\hat{NS}$ and $\hat{IS}$—i.e., $\langle \mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{S}' \rangle$ tuples labeled according to whether they illustrate noninterference or interference—we could train an interpretable machine-learning model and then extract rules to explain to the user what gives rise to interference. A natural such model to consider is a decision tree. In a decision tree, each decision node (i.e., interior node) is a predicate on features of

a $\langle \mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{S}' \rangle$ tuple, and its two children correspond to a true or false evaluation of this predicate on a tuple, respectively. A $\langle \mathsf{C},\ \mathsf{I},\ \mathsf{S},\ \mathsf{S}' \rangle$ tuple is classified by traversing the tree from its root, following the branch from each decision node corresponding to the result of evaluating the predicate at that node on the tuple. Each leaf is labeled with an estimate of the probability that a tuple constrained by the predicates' evaluations from the root to that leaf is in IS. We will discuss what features we include in the process of building decision trees in Sec. 5.2.3, but an example might be individual variables (e.g., $\mathsf{C}(cvar)$).

A single decision tree can easily grow to be deep and complex, and it can miss some useful combinations of predicates since each decision predicate is highly influenced by the splits above it in the tree. To make the decision tree model more powerful in finding useful predicates, we used a decision-tree ensemble called gradient boosted trees [40]. This process produces m trees denoted $T_1, \ldots, T_m$, with associated weights. If we denote by $T_j(\langle \mathsf{C},\ \mathsf{I},\ \mathsf{S},\ \mathsf{S}' \rangle)$ the real number stored at the leaf to which $\langle \mathsf{C},\ \mathsf{I},\ \mathsf{S},\ \mathsf{S}' \rangle$ is assigned by T_j, then the weighted sum of $T_j(\langle \mathsf{C}\ ,\ \mathsf{I},\ \mathsf{S},\ \mathsf{S}' \rangle)$ for $j = 1, \ldots, m$ is an estimate of the probability that $\langle \mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{S}' \rangle \in IS$.

To interpret tree ensembles, rule-based classifiers (e.g., RuleFit [41], Slipper [26], Pre [39]) were introduced to bridge the interpretability of a decision tree with the modeling power of a tree ensemble. Our toolchain leverages Skope-rules[1] to generate logical rules from the tree ensemble. Specifically, consider any path from the root to a leaf in a tree T_j, and let $\pi_{j,1}, \ldots, \pi_{j,\ell}$ denote the predicates along that path that evaluated to true. So, for example, if the first predicate encountered in T_j, say "$\mathsf{C}(cvar) = 1$", evaluated to false, then $\pi_{j,1}$ = "$\mathsf{C}(cvar) \neq 1$". Then, Skope-rules constructs a rule by conjoining $\pi_{j,1}, \ldots, \pi_{j,\ell}$, with the caveat that it limits the number of predicates included in any rule by heuristically pruning them.

Each such rule has an associated *precision* and *recall*, which we evaluate empirically using a validation set held out from $\hat{NS}$ and $\hat{IS}$ during training. That is, the *recall* of a rule is the fraction of validation samples held out from $\hat{IS}$ for which the rule evaluates to true, and the *precision* of the rule is the fraction of validation samples (from $\hat{IS}$ or $\hat{NS}$) for which the rule evaluates to true that were held out from $\hat{IS}$. We further prune rules by iteratively removing conjuncts from a long

[1] `https://skope-rules.readthedocs.io/`

rule if the precision of the resulting rule is at least 95% of the original. We then rank order rules according first to precision, and then according to recall.

As we will show, these rules can assist in a quantitative analysis of the leakage. For example, a procedure may include a backdoor that leaks the secret through a specific attacker-controllable condition (e.g., leaking the secret only when a 64-bit attacker-controlled variable equals 0). The worst-case interference could be masked by a large number of weak interferences, as we mentioned in Sec. 4.1. In that case, the interpretable rule could reveal this specific condition and help the developer re-evaluate the interference under that condition. The case studies in Sec. 5.4.3–5.4.5 illustrate how we choose the attacker-controlled inputs based on the interpretable leakage rules and then re-evaluate the interference caused by a specific side-channel vector.

5.2.3 Feature engineering

The utility of the rule generation described in the previous section depends critically on the features of each $\langle \mathsf{C},\ \mathsf{I},\ \mathsf{S},\ \mathsf{S}' \rangle$ tuple exposed when training the tree ensemble, from which the predicates making up the decision nodes of each tree are formed. One factor that makes feature engineering especially critical here is that the `SAT` solver used to produce elements of $\hat{IS}$ and $\hat{NS}$ requires that the conditions defining IS and NS (i.e., conditions (5.6) and (5.7)) be presented to the `SAT` solver in terms of binary variables only. As such, each solution generated by the `SAT` solver is expressed as an assignment to these binary variables. While for some hardware logic, a binary representation of the relevant variables is most natural, for other types of logic (e.g., on integers), it is not.

For this reason, we augment each binary solution returned by the `SAT` solver (i.e., each $\langle \mathsf{C},\ \mathsf{I},\ \mathsf{S},\ \mathsf{S}' \rangle$ tuple) with additional features. First, we reconstruct features in a type-aware way from their binary representations. For example, if a variable was initially an integer before being reduced to a collection of binary variables in the formula presented to the `SAT` solver, we recover the integer value from the bit-vector solution and include it as a feature on which the tree ensemble can trained. With such type-aware features, predicates such as, e.g., "$\mathsf{S}(svar) < 15$" can be learned in a search for simple predicates testing only a single feature, i.e., unary predicates.

Unary predicates, however, will be unable to naturally capture some relationships resulting in leakage. For example, if leakage happens only when $\mathsf{S}(svar) > \mathsf{C}(cvar)$, permitting only unary predicates will result in a boundary characterized point-by-point, e.g., "$\mathsf{S}(svar) \geq \theta \wedge \mathsf{C}(cvar) < \theta$"

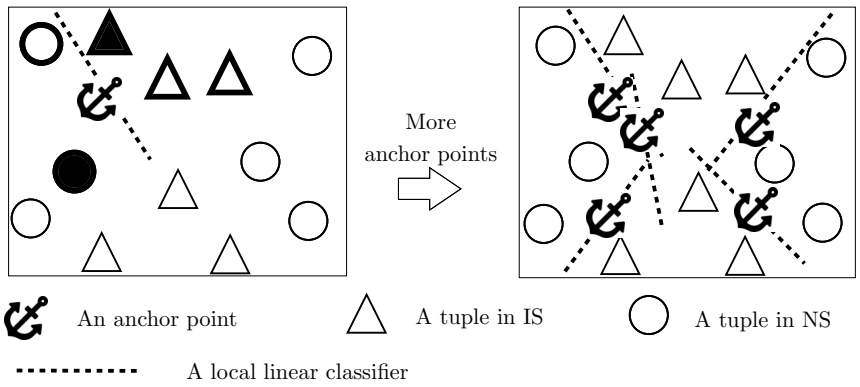


Figure 5.2: Finding linear combinations of features near anchor points

where $\theta = 1, 2, \ldots$ We thus expanded our feature set to permit linear combinations of some features (e.g., "$\mathsf{S}(svar) - \mathsf{C}(cvar)$"), chosen by a linear classifier described below. To accommodate branching in the procedure that results in discontinuities in the boundary between *IS* and *NS*, we opted for a *local* linear classifier (e.g., [36, 82]).

More specifically, to train the classifier with local constraints, we pick *anchor points*, around each of which we train a local classifier that best separates the *nearby* samples in $\hat{IS}$ and $\hat{NS}$. (See Fig. 5.2.) To select anchor points, we first find pairs of $\langle \mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{S}' \rangle$ tuples, one from $\hat{IS}$ and one from $\hat{NS}$, that are *neighbors* in one feature (i.e., after ranking all tuples by this feature, the pair are adjacent in the ranking) and then take the pair's *midpoint* tuple as their per-feature means. We then select anchors uniformly at random from these midpoints. For each anchor, we train a linear classifer using the tuples in $\hat{IS}$ and $\hat{NS}$ that are within a threshold Euclidean distance from the anchor. The linear combination of features used in this linear classifier is then added as another feature to each $\langle \mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{S}' \rangle$ tuple.

5.3 Implementation

We developed our approach for evaluating and interpreting leakage, described in Sec. 5.1 and Sec. 5.2, with an eye toward applying it to evaluate and understand leakage from hardware designs. To do so, we define the procedure *proc* to be a hardware design, say written in Verilog, in its initial state but with a predefined program stored in its memory. Our system, DINoMe, enables the user to annotate the configuration by marking components of the hardware state as attacker-controlled (i.e., in $Vars_{\mathsf{C}}$), attacker-observable (in $Vars_{\mathsf{O}}$), secret (in $Vars_{\mathsf{S}}$), or otherwise unknown to the attacker (in $Vars_{\mathsf{I}}$); to simplify discussion below, we will assume there is one of each, denoted *cvar*, *ovar*, *svar*, and *ivar*, respectively. Our system converts this "procedure," which we continue to

denote *proc*, to a cycle-accurate logical formula Π_{proc} that characterizes hardware execution of the program and that relates C, O, I, and S. The user can also declare a declassification function δ that operates on the hardware state of the system (we will give examples below), from which DINoMe similarly produces a logical formula Π_{δ} that characterizes how the declassified information Δ relates to inputs C, I, and S in the execution of *proc*. From Π_{proc} and Π_{δ}, DINoMe generates $\hat{J}_n^{\delta}$ for varying n (see (5.5)) and, if requested, sample sets $\hat{IS}$ and $\hat{NS}$ from *IS* (see (5.6)) and *NS* (see (5.7)), respectively. These sets seed the generation of the rules for interpreting leakage, as discussed in Sec. 5.2.

In the subsections below, we will discuss particular challenges we encountered when building DINoMe and how we overcame them. We focus on how to extract $\Pi_{proc}(\mathsf{C},\mathsf{I},\mathsf{S},\mathsf{O})$ in Sec. 5.3.1. In Sec. 5.3.2, we describe simplification techniques we leverage as a preprocessing step before performing projected model counting, described in Sec. 5.3.3. Finally, we discuss our technique for sampling to create $\hat{IS}$ and $\hat{NS}$ in Sec. 5.3.4.

5.3.1 Extracting $\Pi_{proc}(\mathsf{C},\mathsf{I},\mathsf{S},\mathsf{O})$

To analyze the leakage from *proc*, we need an accurate postcondition $\Pi_{proc}(\mathsf{C},\mathsf{I},\mathsf{S},\mathsf{O})$ for *proc*. In practice, generating a postcondition for an arbitrary procedure is not trivial. Especially here, where our concern is detecting leakage from a processor implementation when running an application—i.e., the procedure *proc* includes numerous cycles of a cycle-accurate implementation of the processor logic as well as the software logic—the postcondition will be quite large.

Our general strategy to construct $\Pi_{proc}(\mathsf{C},\mathsf{I},\mathsf{S},\mathsf{O})$ in these circumstances is to assemble it one cycle at a time. Yosys [94] provides a framework to convert the Verilog code for a processor design to its internal register-transfer level (RTL) intermediate language, optimize or modify the design using a series of passes, and finally translate the design to targeted formula through its back-end pass. The SMT2 back-end pass defines a data structure for each hardware module representing the module's temporary hardware state, a function to implement the module's state transition from one cycle to the next, and an initialization function to initialize the module's state. To incorporate the software logic of *proc*, we compile the software to its hardware-readable assembly and load the assembly into the instruction memory unit.

To mark the symbolic variables, the analyst defines a configuration file to mark as symbolic each input parameter of *proc* (in this case, *svar*, *ivar*, and *cvar*), which can be a software variable

located at a fixed location in the memory unit or a wire/register inside the hardware module. Our modified YOSYS SMT2 backend pass then tracks the constraints associated with this symbolic data throughout a cycle execution. Specifically, it outputs a logical postcondition $\tau_{proc}(\mathsf{H}^{t-1}, \mathsf{H}^t)$ that relates the hardware state $\mathsf{H}^{t-1} : \mathit{Vars}_{\mathsf{H}} \to \mathit{Vals}_{\mathsf{H}}$ at the end of cycle $t-1$ to the hardware state H^t that results from executing cycle t. We also use it to generate initialization logic $\Psi^0_{proc}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{H}^0)$ that characterizes the first-cycle starting state H^0 using the configured symbolic inputs.

Using the transition logic, we construct a cycle-accurate postcondition Ψ^T_{proc} representing the logic between symbolic inputs and its internal hardware state one cycle at a time, leveraging the entire hardware state as an "observable" output of the cycle.

$$\Psi^T_{proc}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{H}^T) \leftarrow \Psi^0_{proc}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{H}^0) \wedge \bigwedge_{t=1}^{T} \tau_{proc}(\mathsf{H}^{t-1}, \mathsf{H}^t)$$

We finally define $\Pi_{proc}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{O})$ by defining O in terms of the sequence of hardware states $\langle \mathsf{H}^t \rangle^T_{t=0}$ using a formula $\Gamma(\langle \mathsf{H}^t \rangle^T_{t=0}, \mathsf{O})$.

$$\Pi_{proc}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{O}) \leftarrow \Psi^T_{proc}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{H}^T) \wedge \Gamma(\langle \mathsf{H}^t \rangle^T_{t=0}, \mathsf{O}) \tag{5.8}$$

For example, in cache-based side channels, the observable parameters are whether there is a cache hit/miss during the execution, which is constructed using the values of the $s2\_hit$ register across the execution (as demonstrated in Sec. 5.4.3).

In our experiments, we selected T to ensure the termination of the execution, based on our knowledge gained by studying the CPU. A more conservative method would be to track the CPU pipeline and call the `SAT` solver each cycle to check whether the last instruction has certainly committed. We have confirmed that adding more cycles after the termination of the execution does not affect Π_{proc} meaningfully, as the additional cycles do not process any valid opcodes and so only trivially change the hardware state.

5.3.2 Preprocessing formula for #∃SAT

Applying a correct combination of simplification techniques is critical to scaling the sampling of IS and NS to create $\hat{IS}$ and $\hat{NS}$ and to count $\left|\tilde{X}^{\delta}_{S,S'}\right|$ and $\left|\check{X}^{\delta}_{S,S'}\right|$ to compute $\hat{J}^{\delta}_{n}$. Langniez et al. [63] provides a summary of the options. As defined in Sec. 5.1 and Sec. 5.2.1, our computation

task is *projected model counting* (#∃SAT) [6] which counts feasible assignments of selected variables in a propositional formula. The complexity of the counting problem is #NP-hard.

To simplify the logical formula, we use preprocessing (similar to that used in, e.g., [56, 70]) that fully applies equivalence-preserving simplification techniques (e.g., vivification and occurrence reduction), and then partially applies SAT-preserving simplifications (e.g., literal eliminations, variable eliminations and clause eliminations) targeting variables not in our counting target (i.e., not marked as *svar*, *ivar*, *cvar*, or *ovar*). For example, the partially applied blocked-clause elimination will remove a clause if it contains a variable not in the counting target such that every resolvent obtained by resolving on it is a tautology. Especially for hardware designs where the number of possible states increases with the cycle count but many registers are not modified in some cycles, preprocessing the formula can substantially reduce the redundancy between the initial and final cycle formulas.

Although we only need the logic $\Pi_{proc}(\mathsf{C},\mathsf{I},\mathsf{S},\mathsf{O})$ to describe the relationship between the attacker-observable outputs O and inputs C, I, S the translated conjunctive-normal-form (CNF) propositional formula F produced by the commonly used Tseitin algorithm would have numerous auxiliary variables. The number of auxiliary variables and clauses increases quickly with the number of cycles. To reduce the use of auxiliary variables, we applied a state-of-art preprocessing technique for *model counting* called B+E proposed by Lagniez et al. [62], who also discussed a possible application of this method to *projected model counting*. In our modified version of B+E, we partition the variables in the CNF formula representing Π_{proc} into two disjoint variable sets: *Sup* containing the variables in $Vars_{\mathsf{I}}$, $Vars_{\mathsf{C}}$, $Vars_{\mathsf{S}}$, and $Vars_{\mathsf{O}}$, and *Dep* containing all other variables. For each variable v in *Dep* and pair of clauses $\bar{v} \vee C_j$ and $v \vee C'_i$, we then resolve on v and replace the clauses with their resolvent $C_j \vee C'_i$. We do this *only* for variables $v \in Dep$, since this ensures:

$$\left\{ \langle \mathsf{C},\mathsf{I},\mathsf{S},\mathsf{O} \rangle \,\middle|\, \bigwedge_{i=0}^{a} (v \vee C'_i) \wedge \bigwedge_{j=0}^{b} (\bar{v} \vee C_j) \right\} = \left\{ \langle \mathsf{C},\mathsf{I},\mathsf{S},\mathsf{O} \rangle \,\middle|\, \bigwedge_{i=0}^{a} \bigwedge_{j=0}^{b} (C'_i \vee C_j) \right\}$$

Using this algorithm, we eliminate variables in *Dep* so that the final formula uses only *svar*, *ivar*, *cvar*, and *ovar* without any auxiliary variables. This does have the side effect of introducing more complicated clauses, however, and so we avoid eliminating $v \in Dep$ when v is present in numerous clauses (e.g., $b \cdot a > 500$).

	cycles						
	5	10	15	20	25	30	35
No simplification	49.1	126.6	177.0	304.2	459.0	667.6	867.6
Simplified	1.10	1.54	1.98	2.03	89.9	167.3	389.0
#∃SAT-preprocess	1.10	1.41	1.45	1.65	1.71	1.77	1.88

Table 5.1: CNF file size (MB) for logic formulas extracted from the RISC-V BOOM core configured with a small application program, starting from an initial state with some symbolic memory blocks and cache states (see Sec. 5.4.3). The CNF file size for a one-cycle execution with completely symbolic initial state is 40MB. Only computations that terminated within 10 minutes are represented.

In Table 5.1, we present example sizes of CNF formulas generated using different simplification options, for our case studies in Sec. 5.4. The row denoted by "No simplification" represents the size of a directly translated CNF formula from the multi-cycle SMT formula, which linearly increases with the number of cycles. The "Simplified" row is generated by directly applying the CNF simplifications provided by `CryptoMiniSAT 5.0` to the formula in "Original", while the row "#∃SAT-preprocess" is obtained by incrementally applying the preprocessor described in this section to each Ψ^T_{proc} $(\mathsf{C},\mathsf{I},\mathsf{S},\mathsf{H}^T)$ before using it to build Ψ^{T+1}_{proc} $(\mathsf{C},\mathsf{I},\mathsf{S},\mathsf{H}^{T+1})$.

5.3.3 Measurement with declassification using projected model counting

Using `CryptoMiniSAT 5.0` as the basic solver, we implemented a counter to estimate the numerator and the denominator in the measurement $\hat{J}^\delta(S,S')$ in (5.5).

5.3.3(a) Counting $\hat{J}^\delta(S,S')$ To compute the measurement $\hat{J}^\delta(S,S')$, DINOME needs to count the size of $\tilde{X}^\delta_{S,S'}$ and $\check{X}^\delta_{S,S'}$. Directly counting $\tilde{X}^\delta_{S,S'}$ is not easy as the set difference operation will introduce a "forall" quantifier. Fortunately, since $\left|\tilde{X}^\delta_{S,S'}\right| = \left|\check{X}^\delta_{S,S'}\right| - \left|\hat{X}^\delta_{S,S'}\right|$, it suffices to count $\check{X}^\delta_{S,S'}$ and $\hat{X}^\delta_{S,S'}$ for each sample pair S, S'. Intuitively, counting $\check{X}^\delta_{S,S'}$ could be expressed as a projected model counting task over $\langle \mathsf{C},\mathsf{O},\mathsf{I},\mathsf{S}\rangle$ in a quantifier-free SAT problem with two copies of Π_{proc} shown in $\check{F}$ below. $\check{F}$ is translated to a CNF proposition where it uses v bit variables to

represent $\langle \mathsf{C}, \mathsf{O}, \Delta, \mathsf{I} \rangle$ and others to represent $\langle \mathsf{S}, \mathsf{S}', \mathsf{I}, \mathsf{I}' \rangle$ and auxiliary variables.

$$\begin{aligned} \check{F} \leftarrow \quad & \left(\Pi_{proc}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{O}) \vee \Pi_{proc}(\mathsf{C}, \mathsf{I}', \mathsf{S}', \mathsf{O})\right) \\ & \wedge \quad \Pi_{\delta}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \Delta) \wedge \Pi_{\delta}(\mathsf{C}, \mathsf{I}', \mathsf{S}', \Delta) \\ & \wedge \begin{pmatrix} (\mathsf{S}(svar) \in S \wedge \mathsf{S}'(svar) \in S') \\ \vee\ (\mathsf{S}'(svar) \in S \wedge \mathsf{S}(svar) \in S') \end{pmatrix} \end{aligned} \tag{5.9}$$

Following the Sec. 4.2.2(b) , two random, disjoint sets S and S' of expected size n are specified with distinct strings $p, \hat{p} \in \{0,1\}^b$ where $n = |\mathbb{S}| / 2^b$, and specifically with the constraint that for a fixed hash function, the hash of each $s \in S$ is p and the hash of each $s' \in S'$ is $\hat{p}$.

For $\hat{X}^{\delta}_{S,S'}$, we can define another projected model counting [6] task over $\langle \mathsf{C}, \mathsf{O}, \Delta, \mathsf{I} \rangle$ in a quantifier-free SAT problem $\hat{F}$ shown below. $\hat{F}$ uses the logical postcondition Π_{proc} twice, where the first copy is for the execution with a secret $\mathsf{S}(svar) \in S$ and the second checks for existence of a secret $\mathsf{S}'(svar) \in S'$ leading to a result O also possible with S. $\hat{F}$ also checks the existence of some secret (denoted by $\mathsf{S}''(svar)$) in the secret set S' leading to the equivalent declassification value Δ so that we can ensure the S and S' cannot be distinguished by Δ.

$$\begin{aligned} \hat{F} \leftarrow \quad & \Pi_{proc,\delta}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{O}, \Delta) \wedge \mathsf{S}(svar) \in S \\ & \wedge\ \Pi_{proc}(\mathsf{C}, \mathsf{I}', \mathsf{S}', \mathsf{O}) \wedge \mathsf{S}'(svar) \in S' \\ & \wedge\ \Pi_{\delta}(\mathsf{C}, \mathsf{I}'', \mathsf{S}'', \Delta) \wedge \mathsf{S}''(svar) \in S' \end{aligned} \tag{5.10}$$

5.3.3(b) Optimizations for counting $\tilde{X}^{\delta}_{S,S'}$ and $\check{X}^{\delta}_{S,S'}$ Enumerating all solutions to (5.9) and (5.10) using a solver is intractable. To estimate the number of solutions to each instead, we used the approximate model counting technique due to Chakraborty et al. [15], specifically the approach taken by Soos and Meel [88].

That is, by specifying a randomly selected hash function $\hat{H}^{\hat{b}} : \{0,1\}^v \to \{0,1\}^{\hat{b}}$ and an output $\hat{p} \in \{0,1\}^b$ as an additional constraint, we can estimate $\left|\hat{X}^{\delta}_{S,S'}\right|$ using the average value of multiple estimations of $\left|\hat{Z}^{\hat{p}}_{S,S'}\right|$ with some error ϵ and confidence δ (i.e., $\left|\hat{X}^{\delta}_{S,S'}\right| \approx \left|\hat{Z}^{\hat{p}}_{S,S'}\right| \times 2^{\hat{b}}$). Similarly, we could estimate $\left|\check{X}^{\delta}_{S,S'}\right|$ using $\check{Z}^{\check{p}}_{S,S'}$.

$$\check{Z}^{\hat{p}}_{S,S'} = \left\{ \langle \mathsf{C}, \mathsf{O}, \Delta, \mathsf{I} \rangle \;\middle|\; \langle \mathsf{C}, \mathsf{O}, \Delta, \mathsf{I} \rangle \in \check{X}^{\delta}_{S,S'} \wedge \; \check{H}^{\check{b}}(\langle \mathsf{C}, \mathsf{O}, \Delta, \mathsf{I} \rangle) = \check{p} \right\} \tag{5.11}$$

$$\hat{Z}^{\hat{p}}_{S,S'} = \left\{ \langle \mathsf{C}, \mathsf{O}, \Delta, \mathsf{I} \rangle \;\middle|\; \langle \mathsf{C}, \mathsf{O}, \Delta, \mathsf{I} \rangle \in \hat{X}^{\delta}_{S,S'} \wedge \; \hat{H}^{\hat{b}}(\langle \mathsf{C}, \mathsf{O}, \Delta, \mathsf{I} \rangle) = \hat{p} \right\} \tag{5.12}$$

This optimization for model counting will limit the number of calls to the SAT solver by constraining the number of solutions available, and thus make the counting more scalable for large set size. Thus, $\hat{J}^{\delta}\ (S,\ S')$ is estimated using the average value of $1 - \frac{\left|\hat{Z}^{\hat{p}}_{S,S'}\right|}{\left|\check{Z}^{\check{p}}_{S,S'}\right|}$ for various $\hat{p}$, $\check{p}$.

Our primary departure from the implementation by Soos and Meel [88] lies in utilizing task-specific properties in our counting tasks to reduce redundant effort in solution searching. Specifically, since $\hat{X}^{\delta}_{S,S'} \subseteq \check{X}^{\delta}_{S,S'}$, we ensure that $\hat{X}^{\delta}_{S,S'} \cap \check{Z}^{\check{p}}_{S,S'} \subseteq \hat{Z}^{\hat{p}}_{S,S'}$ in our counting by defining $\hat{H}^{\hat{b}}(\langle \mathsf{C}, \mathsf{O}, \Delta, \mathsf{I} \rangle)$ to be the $\hat{b}$-bit prefix of $\check{H}^{\check{b}}(\langle \mathsf{C}, \mathsf{O}, \Delta, \mathsf{I} \rangle)$ for $\hat{b} \leq \check{b}$. Then once we have generated solutions in $\check{Z}^{\check{p}}_{S,S'}$, we speed up finding solutions in $\hat{Z}^{\hat{p}}_{S,S'}$ for $\hat{b} = \check{b}$ (and so $\hat{p} = \check{p}$) by first checking each solution in $\check{Z}^{\check{p}}_{S,S'}$ to see if it satisfies $\hat{F}$ (i.e., is in $\hat{X}^{\delta}_{S,S'} \cap \check{Z}^{\check{p}}_{S,S'}$). Only if insufficient solutions are found with $\hat{b} = \check{b}$ is $\hat{b}$ reduced and the solver used to generate additional solutions in $\hat{Z}^{\hat{p}}_{S,S'}$ for $\hat{p}$ a $\hat{b}$-bit prefix of $\check{p}$.

In the case studies in Sec. 5.4, we set the error $\epsilon = 0.4$ and confidence parameter $\delta = 0.9$ in this method of estimating the sizes of $\tilde{X}^{\delta}_{S,S'}$ and $\check{X}^{\delta}_{S,S'}$, from which $\hat{J}^{\delta}(S, S')$ is estimated using (5.5). For each set size n, we compute $\hat{J}^{\delta}_{n}$ using at least 100 hash functions, i.e., implicit selection of pairs S, S' of expected size n.

5.3.4 Sampling $\hat{NS}$ and $\hat{IS}$ for interpretable learning

Similar to the counting process, to construct $\hat{NS}$ and $\hat{IS}$, the sampler will select hash functions H randomly from a family and output values p randomly from its range to solve for tuples $\langle \mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{S}' \rangle$ for which $H(\langle \mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{S}' \rangle) = p$ (and are in NS or IS, respectively). In the following experiments, we will generate up to 100,000 solutions for each of $\hat{NS}$ and $\hat{IS}$, where 70% used for training and 30% used for validation.

We cannot directly encode set difference, used in (5.6) and (5.7), using an equivalent quantifier-free formula. To implement a sampler to generate solutions in the set difference, we will use one solver ("E-solver") to search for candidate solutions and another ("F-solver") cancel candidates; this is a commonly used algorithm for an SMT solver to solve exist-forall problems (e.g., see [32]).

E-Solver with H and p generates $\langle \mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{S}', \mathsf{O}, \Delta \rangle$ satisfying

$$\Pi_{proc,\delta}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{O}, \Delta) \wedge \Pi_{proc,\delta}(\mathsf{C}, \mathsf{I}', \mathsf{S}', \mathsf{O}', \Delta) \wedge \mathsf{O} \neq \mathsf{O}' \wedge H(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{S}') = p \tag{5.13}$$

F-Solver cancels $\langle \mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{S}', \mathsf{O}, \Delta \rangle$ satisfying (5.13) if there is some I'' satisfying

$$\Pi_{proc,\delta}(\mathsf{C}, \mathsf{I}'', \mathsf{S}', \mathsf{O}, \Delta) \tag{5.14}$$

Figure 5.3: Generating examples in $\hat{IS}$ using EF-solver

Here, we will illustrate sampling *IS*, while sampling *NS* is similar.

Specifically, the sampler first uses the E-Solver to generate feasible solutions $\langle \mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{S}' \rangle$ (see (5.13)) that guarantee, for an attacker's choosen C, the observable value O derived from S with I could be different from an observable O' generated by S' with some I' when the declassified value Δ is the same. However, it does not guarantee the O is never feasible for S. To further test whether the $\langle \mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{S}' \rangle$ is in $\hat{IS}$, we use the F-Solver to test whether $\langle \mathsf{I}'', \mathsf{S}' \rangle$ for some I'' could generate O with $\langle \mathsf{I}, \mathsf{S} \rangle$ when they share the declassification value Δ, to check whether we need to cancel the solution. That is, $\langle \mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{S}' \rangle$ satisfying (5.13) but not (5.14) will be included in $\hat{IS}$.

After generating enough $\langle \mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{S}' \rangle$ tuples in $\hat{NS}$ and $\hat{IS}$, the interpretation module trains local support vector machine (SVM) classifiers [37] around each of 50 anchor points, after ruling out data whose normalized Euclidean distance (i.e., after scaling each attribute to a value between 0 and 1, use Euclidean distance divided by the number of attributes) is more than 0.2 from the anchor. Then a logistic regression model for *NS* and *IS* is learned using a gradient boosted tree implementation *xgboost* [19]. To generate the interpretable models, we implemented the rule learner using Skope-rules.

5.4 Case Studies

In this section, we illustrate DINoMe by describing its application to the BOOM core (`https://github.com/riscv-boom/riscv-boom`), an open-source RISC-V core that is susceptible to cache-based side channels and Spectre attacks. The goal of these case studies was to illustrate our methodology and to show how it can be useful to system analysts. We require these analysts to specify the secret to protect and attacker-controlled and attacker-observable variables but, critically, not the specific attacker algorithm.

- We applied DINoMe to evaluate cache-based side-channel leakage due to secret-dependent mem-

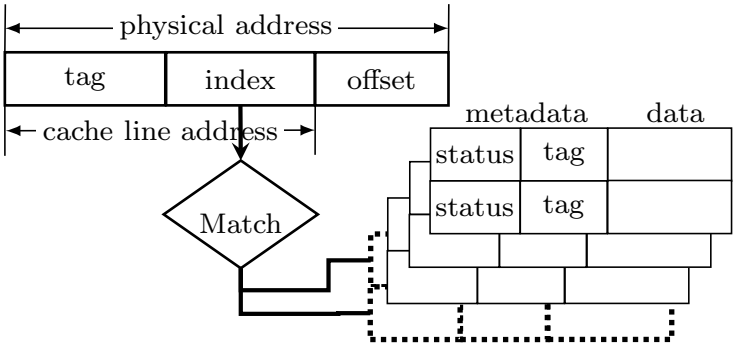


Figure 5.4: Way-associated cache in BOOM

ory accesses. With different parameter configurations for BOOM (i.e., number of cache ways w and whether to enable memory sharing), this case study shows how $\hat{J}_n^\delta$ curves demonstrate the effects of these settings on the leakage. Moreover, we implemented and evaluated two possible mitigations, namely ScatterCache [93] and PhantomCache [90], both of which use a per-security-domain memory-to-cache mapping to reduce but not eliminate the cache leakage. Our measurements using $\hat{J}_n^\delta$ illustrate which mitigation is better for a specific BOOM setting.

- We used DINoMe to demonstrate information leakage due to cache-based side channels from a modular exponentiation function commonly used in cryptographic algorithms. The rule-based interpretation explains how to choose attacker-controlled variables and which portion of the secret are leaked.
- We evaluated software code snippets causing speculative execution and demonstrated how to use declassification to narrow in on leakage caused by speculative execution (i.e., by declassifying other leakage to reveal it). We found that some software with a short speculative window is not sufficient to cause out-of-bound memory leakage in the latest version of BOOM.

5.4.1 BOOM configurations

BOOM provides a configurable L1 cache module using a random replacement policy, where its memory-to-cache mapping is shown in Fig. 5.4. In the following experiments, we used pocket-size hardware modules to replace the modules in the BOOM v2.2.3 configuration. Although the system we evaluated is configured to be much smaller than an actual system, it preserves all of the original functionality; analyzing artificially small but otherwise faithful configurations of a system is not uncommon on model checking, for example (e.g., [8, 2]). Specifically, we set the cache line size to $bbytes = 64$B and the total L1 data cache size to 1KB (16 cache lines in total). We then varied the

cache ways w and sets c (i.e., subject to $w \times c = 16$) in Sec. 5.4.3 but used a fixed setting $c = 2$, $c = 8$ for other evaluations. For the main memory, we set the memory size to 4KB and thus a memory address is only 12 bits. For evaluation purposes, we used the upper half of the memory address space as instruction memory and the lower half as data memory. To simplify the following analysis, we removed the page table walker (PTW) module and assumed virtual addresses were the same as physical addresses. For the instruction fetch, we set the fetch width to 4 and configured the L1 instruction cache to a 1KB, 8-set, 2-way cache with a customized prefetching module that preloaded the software workload at the first cycle.

One feature of BOOM is that it supports speculative execution, with which we will experiment in Sec. 5.4.6. Speculative execution leverages a *branch predictor*, for which we used the GShare branch predictor. The logical structure of GShare is shown in Fig. 5.5. When a prediction request arrives for a branch instruction, the GShare predictor derives a value *bidx* from the certain bits (denoted 'idx' in Fig. 5.5) in the instruction address and an instruction history register and then uses *bidx* to index into a table to which we refer as 'bpd'. Each entry of the 'bpd' table includes a label called 'CFI' and a 2-bit 'state', of which one bit indicates whether the entry holds a strong or weak prediction and the other bit holds that prediction (i.e., whether the branch will be taken or not). If the 'bpd{*bidx*}.CFI' value matches the 'CFI' portion of the instruction address, then the predictor uses the 'bpd{*bidx*}.state' value to make a branch prediction. The GShare predictor will globally tune entries based on executions in any user's domain. Thus, an attacker can easily affect the 'bpd' table before victim's execution, and so we include 'bpd' in $Vars_{\mathsf{C}}$. In our evaluation, we fix the number of 'bpd' entries to 4 so that only 2 bits in the instruction address are used as 'idx' while another 2 bits (=$\log_2$(fetch width)) are used as its 'CFI' label.

In the following case studies, we added the 'bpd' table in the GShare module to $Vars_{\mathsf{C}}$ and registers in the L1 data cache module including the cache metadata, the replacement state (i.e., the linear-feedback shift register (LFSR) for the random replacement policy), and the memory-to-cache mapping (if using a nonfixed mapping) to $Vars_{\mathsf{I}}$.

5.4.2 Logically modeling cache states

The most common cache-based side-channel attacks are Prime+Probe, Flush+Reload, and their variants (e.g., see [98, 95]). In a Prime+Probe attack, the attacker loads memory blocks to fill (Prime) cache sets, permits the victim computation to run for a Prime+Probe interval, and

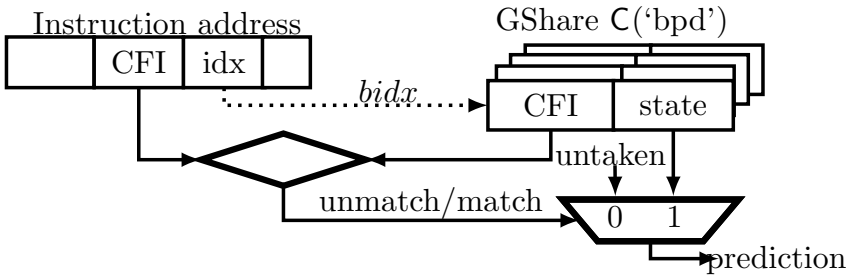


Figure 5.5: Logical architecture for GShare branch predictor

then reads (Probes) these same blocks to determine which were evicted by the victim computation during the Prime+Probe interval. In a Flush+Reload attack, the attacker Flushes a shared-memory block from cache and then, after a Flush+Reload interval, accesses (Reloads) the block to determine whether the block was brought back into the cache by the victim computation.

To model these attacks in our framework, it is necessary to model the effects on the cache of the phases before victim execution (the Prime and Flush steps) and to define O to include the results of the phases after victim execution (the Probe and Reload steps). To do so, we assume that the adversary has access to memory blocks $block_1$, $block_2$, ..., $block_m$ aligned to cache lines, and we define the RISC-V assembly routine *acc* by which the adversary can access the block with index $\ell = \hat{\mathsf{C}}$('blockIdx') and empty $\hat{\mathsf{S}}$:

$$
\begin{aligned}
&acc\ (\hat{\mathsf{C}}, \hat{\mathsf{I}}, \hat{\mathsf{S}})\\
&\quad \texttt{li s0, 0x2000000}\\
&\quad \texttt{add s1, s0, } \ell\\
&\quad \texttt{sll s1, s1, 6}\\
&\quad \texttt{lbu a2, 0(s1)}
\end{aligned}
$$

Starting from hardware state $\hat{\mathsf{H}}^0_\ell$ ($= \hat{\mathsf{I}}$) that is completely symbolic, we generate the per-cycle logical postcondition $\tau_{acc}(\hat{\mathsf{H}}^{t-1}_\ell, \hat{\mathsf{H}}^t_\ell)$ for each $0 < t \leq \hat{T}$ as in Sec. 5.3.1, where we empirically choose $\hat{T} = 45$.

We use these postconditions in two ways. First, we use them to extract a constraint $\Gamma(\langle \mathsf{H}^t \rangle_{t=1}^T, \mathsf{O})$ that defines the attacker's observations O in terms of the hardware states $\langle \mathsf{H}^t \rangle_{t=1}^T$ induced by the execution (see (5.8)). A naive attempt to do so would be to simply include in O the metadata for each cache line at every step of the execution. However, this would grant too much power to an attacker, who should not be given access to the tag values and the exact locations of blocks inside

a set. Instead, we permit only a weaker attacker (cf., abstract noninterference [42]) by defining the constraint $\Gamma(\langle \mathsf{H}^t \rangle_{t=1}^{T}, \mathsf{O})$ that represents the view of cache hits and misses immediately observable by the adversary, by:

$$\mathsf{O}(\text{`hit'})[\ell] = \begin{pmatrix} (\hat{\mathsf{H}}_\ell^0 = \mathsf{H}^T) \wedge \left(\bigwedge_{t=1}^{\hat{T}} \tau_{acc}(\hat{\mathsf{H}}_\ell^{t-1}, \hat{\mathsf{H}}_\ell^t)\right) \\ \wedge \ \left(1 - \bigvee_{t=1}^{\hat{T}} \textsc{CacheMiss}(\hat{\mathsf{H}}_\ell^t, \mathit{block}_\ell)\right) \end{pmatrix}$$

for $\ell = \hat{\mathsf{C}}(\text{`blockIdx'})$. Here, CacheMiss is a BOOM-defined Verilog code snippet that, intuitively, checks a set of cache lines where $\mathit{block}_\ell$ might reside and returns 1 (in a register called `s2_hits`) if none of those cache lines has a valid tag matched with block_ℓ (and returns 0 otherwise). In this way, we characterize the procedure *acc* using a logical postcondition without manually modeling CacheMiss.

Second, we permit the attacker to control which of its blocks are loaded into the cache before the victim runs. Specifically, the predicate $\Psi^0_{proc}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{H}^0)$ that controls the initial hardware state from which the victim executes is modified to constrain which of the attacker's blocks are present in cache, as communicated through a reserved variable 'load' $\in \mathit{Vars}_{\mathsf{C}}$, for which the $\mathsf{C}(\text{`load'})$ is a bit vector of length $m$. That is, attacker block $\mathit{block}_\ell$ should be loaded before the victim runs if and only if $\mathsf{C}(\text{`load'})[\ell] = 1$. To effect this in the $\Psi^0_{proc}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{H}^0)$, we construct $\Psi^0_{proc}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{H}^0)$ to include

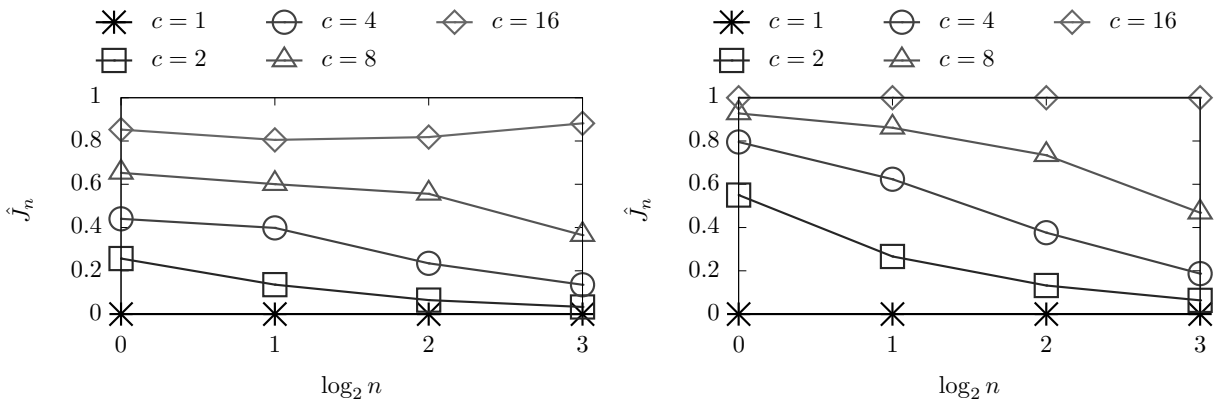
$$\mathsf{C}(\text{`load'})[\ell] = \begin{pmatrix} (\hat{\mathsf{H}}_\ell^0 = \mathsf{H}^0) \wedge \left(\bigwedge_{t=1}^{\hat{T}} \tau_{acc}(\hat{\mathsf{H}}_\ell^{t-1}, \hat{\mathsf{H}}_\ell^t)\right) \\ \wedge \ \left(1 - \bigvee_{t=1}^{\hat{T}} \textsc{CacheMiss}(\hat{\mathsf{H}}_\ell^t, \mathit{block}_\ell)\right) \end{pmatrix}$$

Of course, we rename variables to ensure no conflicts between copies of $\hat{\mathsf{H}}_\ell^t$ included within the $\mathsf{C}(\text{`load'})[\ell]$ and $\mathsf{O}(\text{`hit'})[\ell]$ constraints.

For an attacker, it is sufficient to control the cache using $m = 16$ blocks as the L1 data cache consists of only 16 cache lines in our experiments.

5.4.3 Cache-based side channels

In this section, we evaluate cache-based side channels under different memory isolation and cache configurations.



(a) Symbolic C('load') (b) $\forall \ell : \mathsf{C}(\text{'load'})[\ell] = 1$

Figure 5.6: $\hat{J}_n$ for PRIME+PROBE attacks

5.4.3(a) Without shared memory Here, we target a victim's RISC-V assembly *proc* to access a secret-indexed memory block not shared with the attacker, by setting the base address in `s0` to a value `0x2000010`, in contrast to the one used in *acc*.

$$
\begin{array}{l}
\mathit{proc}\ (\mathsf{C}, \mathsf{I}, \mathsf{S}) \\
\quad \texttt{li s0, 0x2000010} \\
\quad \texttt{add s1, s0, } \mathsf{S}(\text{'secret'}) \\
\quad \texttt{sll s1, s1, 6} \\
\quad \texttt{lbu a2, 0(s1)}
\end{array}
\tag{5.15}
$$

We experimented with different numbers of cache sets c including $c = 1$ (i.e., 1-way, 16-set, fully associative), $c = 2$ (i.e., 8-way, 2-set), $c = 4$ (i.e., 4-way, 4-set), $c = 8$ (2-way, 8-set), and $c = 16$ (i.e., 1-way, 16-set, direct-mapped). As shown in Fig. 5.6(a), $\hat{J}_n$ increases when the number of sets increases. Specifically, there is no leakage ($\hat{J}_n = 0$ for all n) when $c = 1$. Using fewer cache sets, each cache set is shared by more memory blocks, and so an attacker will have more difficulty distinguishing one execution from others. When $1 < c < 16$, $\hat{J}_n$ decreases as n grows, since the attacker can learn only $log_2(c)$ bits about the secret and thus may be unable to distinguish secrets in large sets (i.e., large n).

An example *interference* rule for *IS* generated as described in Sec. 5.2 with the highest precision

(1.00) and a recall ≈ 0.04 in a 2-way, 8-set cache is:

$$\begin{aligned} & \mathsf{S}(\text{`secret'})[2] \geq 1 \quad \wedge \quad \mathsf{S}(\text{`secret'})[1] < 1 \\ \wedge \quad & \mathsf{S}(\text{`secret'})[0] \geq 1 \quad \wedge \quad \mathsf{S}'(\text{`secret'})[1] \geq 1 \\ \wedge \quad & \mathsf{C}(\text{`load'})[5] \geq 1 \quad \wedge \quad \mathsf{C}(\text{`load'})[13] \geq 1 \end{aligned} \tag{5.16}$$

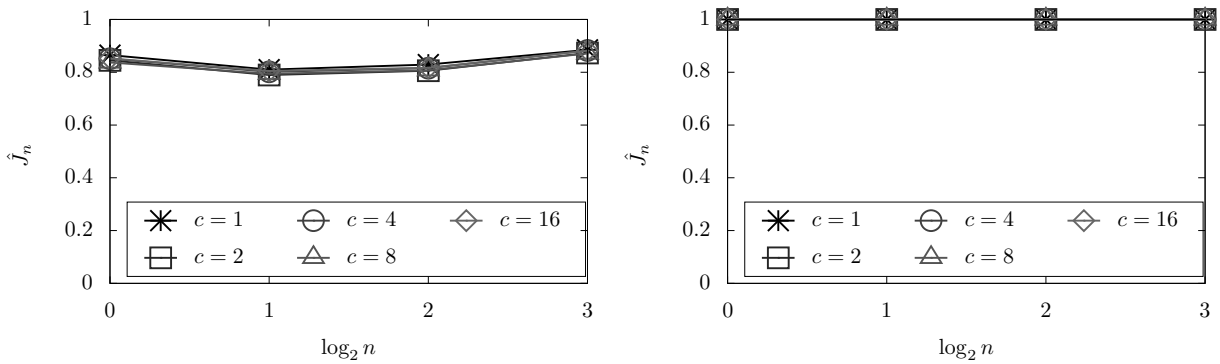
In this rule, the S and S' conjuncts concretize the least significant 3 bits of $\mathsf{S}(\text{`secret'})$ (i.e., $\mathsf{S}(\text{`secret'}) \equiv 5 \bmod 8$) and the lowest bit of $\mathsf{S}'(\text{`secret'})$ (i.e., $\mathsf{S}'(\text{`secret'}) \equiv 0 \bmod 2$). The C conjuncts are $\mathsf{C}(\text{`load'})[5] \geq 1$ and $\mathsf{C}(\text{`load'})[13] \geq 1$; note that $13 \equiv 5 \bmod 8$. That is, an attacker could load all blocks $block_\ell$ with $\ell \equiv 5 \bmod 8$ into cache to distinguish a secret $\mathsf{S}(\text{`secret'}) \equiv 5 \bmod 8$ from $\mathsf{S}'(\text{`secret'}) \bmod 8 \in \{0, 2, 4, 6\}$.

Our approach could not directly represent $\mathsf{C}(\text{`load'})[\ell] \equiv \mathsf{S}(\text{`secret'}) \bmod c$. So, the trees in the model split the dataset based on the cache set index. As such, there were many other top-ranking rules similar to (5.16), each focusing on one residue class of the secret value modulo c where $c = 8$ and constraining $\mathsf{C}(\text{`load'})[\ell] = 1$ for all ℓ with that residue class modulo c. Each such rule works for $\frac{1}{8}$ of S's domain and $\frac{1}{2}$ of S''s domain, thus only for $\frac{1}{8} \times \frac{1}{2} \approx 0.06$ of secret pairs. The recall rate $0.04 < 0.06$ indicates that priming the corresponding cache set ensures (i.e., precision $= 1.0$) the interference but is not necessary to cause it.

Analogously, we can generate rules for the *noninterference* set *NS*, as well. One example with precision 1.0 (i.e., that ensures noninterference) and recall 0.11 constrains the secret's least-significant 3 bits to be the same for S and S':

$$\begin{aligned} & |\mathsf{S}(\text{`secret'})[2] - \mathsf{S}'(\text{`secret'})[2]| < 1 \\ \wedge \quad & |\mathsf{S}(\text{`secret'})[1] - \mathsf{S}'(\text{`secret'})[1]| < 1 \\ \wedge \quad & |\mathsf{S}(\text{`secret'})[0] - \mathsf{S}'(\text{`secret'})[0]| < 1 \end{aligned} \tag{5.17}$$

This analysis illustrates that an attacker can easily distinguish $\mathsf{S}(\text{`secret'})$ and $\mathsf{S}'(\text{`secret'})$ when priming a cache set used by $\mathsf{S}(\text{`secret'})$ or $\mathsf{S}'(\text{`secret'})$ but not both. It is therefore safe to assume that the attacker will PRIME the cache using all its controlled memory blocks to maximize the



(a) Symbolic C('load') (b) $\forall \ell : \mathsf{C}(\text{'load'})[\ell] = 0$

Figure 5.7: $\hat{J}_n$ for FLUSH+RELOAD attacks

chances for leakage. The $\hat{J}_n$ measure under this specific attack is shown in Fig. 5.6(b). The worst case will leak all of the 4-bit secret when using high-granularity memory-to-cache mapping, i.e., where $c = 16$.

5.4.3(b) With shared memory To evaluate the leakage with memory sharing enabled (i.e., with FLUSH +RELOAD attacks), we allow the attacker to control and observe all memory blocks used by the victim by setting the base to `0x2000000` in *proc* instead of to `0x2000010` (see (5.15)). Fig. 5.7(a) shows the corresponding $\hat{J}_n$. The $\hat{J}_n$ curves are similar and close to 1 for all settings, indicating that the leakage does not have much correlation with w. An example rule for interference derived using the methodology of Sec. 5.2, having a precision of 1.0 and recall of ≈ 0.04, is

$$\mathsf{S}'(\text{'secret'}) < 2 \wedge \mathsf{S}'(\text{'secret'}) \geq 1 \wedge \mathsf{C}(\text{'load'})[1] < 1 \tag{5.18}$$

That is, if $\mathsf{S}'(\text{'secret'}) = 1$ then $\mathsf{C}(\text{'load'})[1] = 0$ results in interference. Indeed, the other top-ranked rules for this example (not shown) were roughly 32 similar rules, each one setting $\mathsf{C}(\text{'load'})[\ell] = 0$ for a specific secret value $\mathsf{S}(\text{'secret'}) = \ell$ or $\mathsf{S}'(\text{'secret'}) = \ell$. The intuition behind these rules is that an attacker can precisely detect if $\mathsf{S}(\text{'secret'}) = \ell$ by setting $\mathsf{C}(\text{'load'})[\ell] = 0$ (i.e., FLUSHing $block_\ell$ so he can later RELOAD it), and similarly for $\mathsf{S}'(\text{'secret'})$. Going further, if an attacker sets $\mathsf{C}(\text{'load'})[\ell] = 0$ for all ℓ, he can detect the victim's access to any $block_\ell$, as shown in Fig. 5.7(b) where $\hat{J}_n = 1$ for all n.

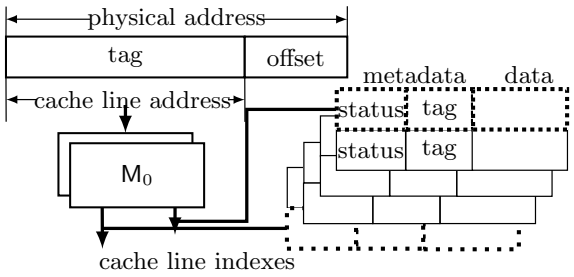


Figure 5.8: ScatterCache

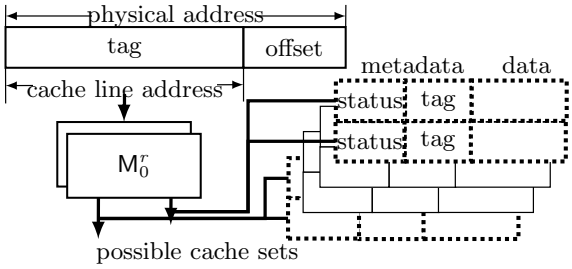


Figure 5.9: PhantomCache (r =2)

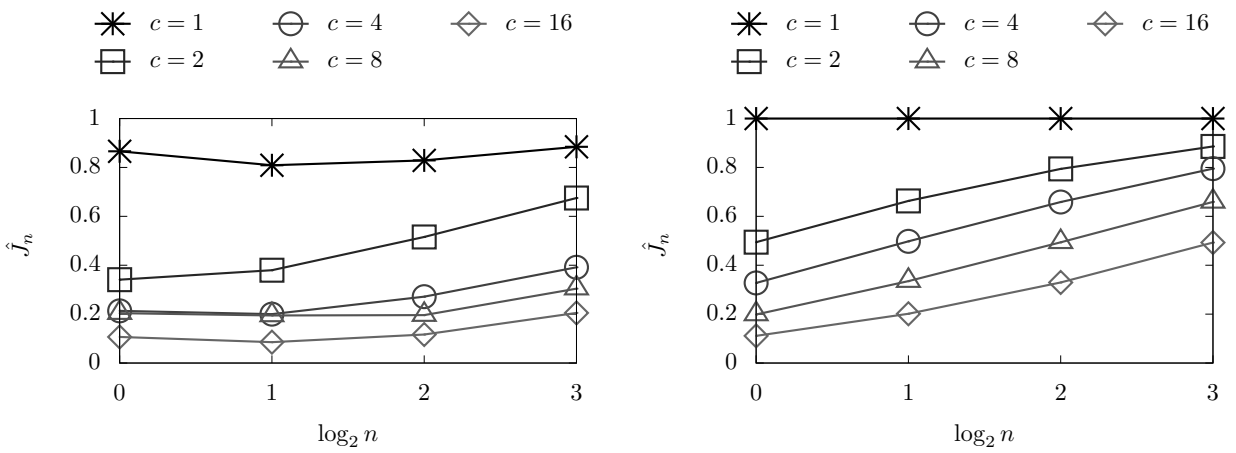
5.4.4 Side-channel-resistant cache designs

To demonstrate the power of DINoMe in comparing different implementations, we evaluate two cache designs for mitigating side channels, namely ScatterCache [93] and PhantomCache [90]. Unfortunately, Verilog specifications of these are unavailable, and so we implemented two simplified cache modules (which we continue to refer to as ScatterCache and PhantomCache) in BOOM following their paper designs.

ScatterCache maps a memory block to a cache line using a cryptographic index derivation function computed using the block's physical address and a private key. To simulate this index derivation without choosing a concrete function, in Fig. 5.8, we use a symbolic look-up table denoted by M_{dom} per security domain dom ($dom = 0$ denotes the victim's domain and $dom = 1$ denotes the attacker's) to store the mapping from memory address to cache line. For security domain dom, its access to memory contents at physical address $paddr$ and so with block address $baddr = \lfloor paddr/bbytes \rfloor$ is mapped to cache lines with way index k and set index $j = \mathsf{M}_{dom}\{baddr\}\{k\}$ for $k = 0, 1, \ldots, w-1$. Similarly, for PhantomCache, we used a domain-specific memory-to-cache mapping represented by M^r_{dom} to allow a memory block to use cache lines in *up to* r cache sets indexed by $\mathsf{M}^r_{dom}\{baddr\}\{k\}$ for $k = 0, 1, \ldots, r$.[2] In the following evaluation, we have $\mathsf{M}_{dom}, \mathsf{M}^r_{dom} \in Vars_{\mathsf{I}}$.

5.4.4(a) Random memory-to-cache mappings First, we experimented without memory sharing when assuming the memory-to-cache mapping is completely unknown to the attacker. We

[2]In contrast to the original paper [90], we do not force each memory block to map to r *unique* cache sets, i.e., we do not constrain $\mathsf{M}^r_{dom}\{baddr\}\{k\} \neq \mathsf{M}^r_{dom}\{baddr\}\{k'\}$ for $k \neq k'$.



(a) Symbolic C('load') (b) $\forall \ell : \mathsf{C}(\text{'load'})[\ell] = 0$

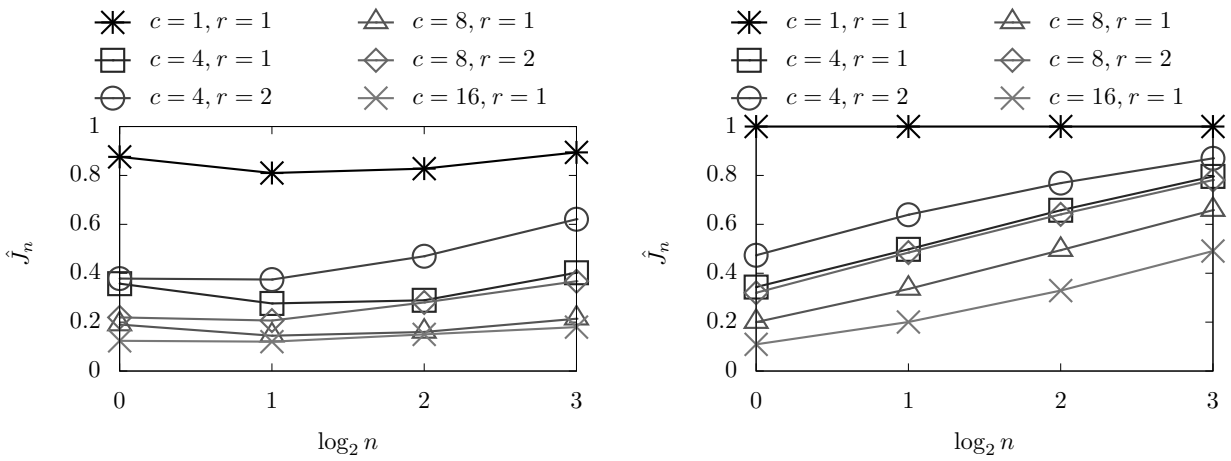
Figure 5.10: ScatterCache, unknown M_{dom}, memory sharing enabled (Flush+Reload attack)

ended up with $\hat{J}_n = 0$ for all n in both ScatterCache and PhantomCache. The attacker cannot tell which memory blocks are accessed by the victim, as an memory block could be mapped to any cache line if the mapping is unknown. However, with shared memory, shown in Fig. 5.10(a) and Fig. 5.10(b), it is still possible to learn some information about which memory block is accessed by the victim. $\hat{J}_n$ is high when n is large, indicating the attacker can precisely determine S('secret') when leakage occurs. Our results indicate that lower cache set granularity leaks more: In Fig. 5.10(a), $c = 1$ leaks the most, which is similar to the normal cache. When $c > 1$, the leakage is reduced.

Overall, with same cache set granularity, $\hat{J}_n$ is higher with PhantomCache with $r = 2$ than PhantomCache with $r = 1$ and ScatterCache when memory is shared. This is because setting $r = 2$ allows one physical address to be mapped to more cache sets and so gains more chance to share cache lines cross domains.

Intuitively, Flush+Reload is the best attacker strategy for a normal cache design when memory sharing is enabled. However, for a new cache design, it may not be clear that it is still the best. Our leakage rules provide some insight for ScatterCache and PhantomCache. For example, two top-ranking rules for ScatterCache, both with precision ≥ 0.80 and recall of ≈ 0.02, are:

$$\begin{aligned} &\mathsf{S}(\text{'secret'})[3] \geq 1 \quad \wedge \quad \mathsf{S}(\text{'secret'})[2] < 1 \wedge \quad \mathsf{S}(\text{'secret'})[1] < 1 \quad \wedge \mathsf{S}(\text{'secret'})[0] < 1 \\ \wedge \quad &\mathsf{I}(\mathsf{M}_0\{8\}\{1\}) \geq 5 \quad \wedge \quad \mathsf{I}(\mathsf{M}_1\{8\}\{1\}) \geq 5 \wedge \quad \mathsf{C}(\text{'load'})[8] < 1 \end{aligned} \tag{5.19}$$



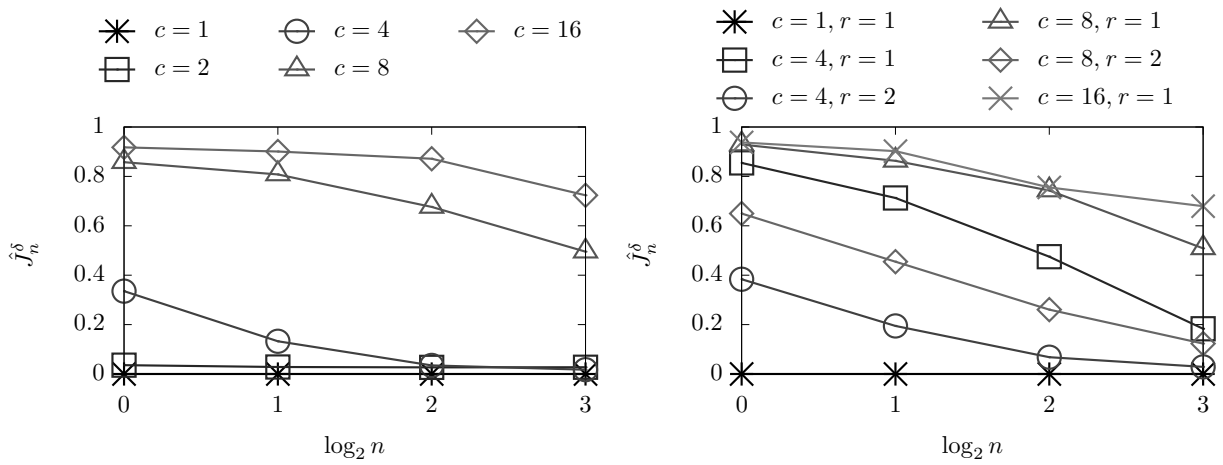
(a) Symbolic C('load') (b) $\forall \ell : \mathsf{C}(\text{'load'})[\ell] = 0$

Figure 5.11: PHANTOMCACHE, unknown M^r_{dom}, memory sharing enabled (FLUSH+RELOAD attack)

These rules are similar to (5.18) but with some additional predicates about M_0. Specifically, (5.19) adds $\mathsf{I}(\mathsf{M}_0\{8\}\{1\}) \geq 5 \wedge \mathsf{I}(\mathsf{M}_1\{8\}\{1\}) \geq 5$ to the rule when setting $\mathsf{C}(\text{'load'})[8] = 0$ (i.e., attacker FLUSHes $block_8$) and $\mathsf{S}(\text{'secret'}) = 8$, which indicates that the $block_8$ should occupy line $k = 1$ in set $j = 5$ in both the victim's and attacker's domains to ensure leakage about whether $\mathsf{S}(\text{'secret'}) = 8$ when the attacker RELOADs $block_8$.

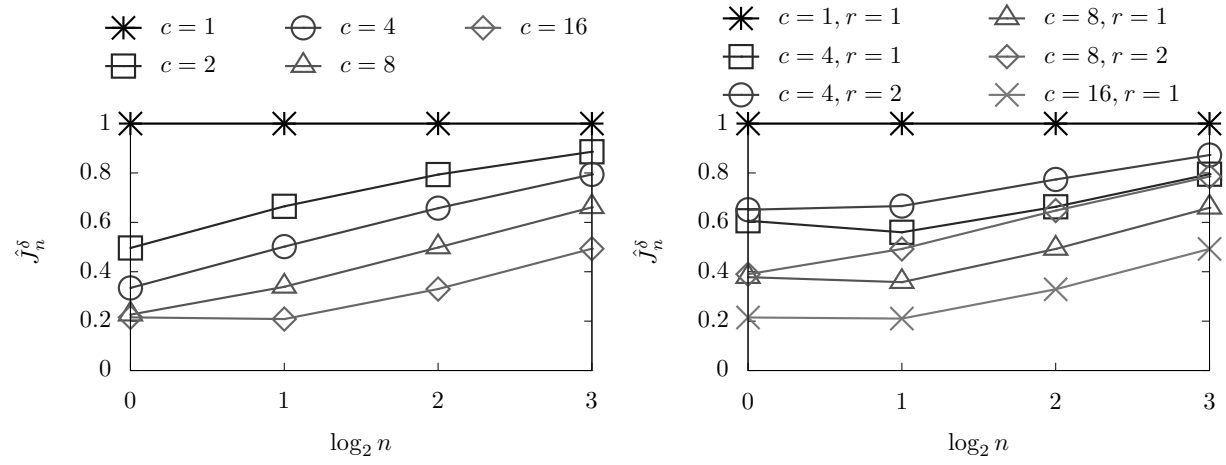
Thus, an attacker should FLUSH+RELOAD all blocks that could share cache lines between victim's and attacker's domain to cause more leakage. Since the memory-to-cache mapping is unknown, an attacker may FLUSH+RELOAD all shared memory blocks. The resulting $\hat{J}_n$ is shown in Fig. 5.10(b) for SCATTERCACHE and Fig. 5.11(b) for PHANTOMCACHE. Under the equivalent cache settings, $\hat{J}_n$ is higher when the attacker takes maximum advantage of FLUSH+RELOAD attacks (versus not, shown in Fig. 5.10(a) and Fig. 5.11(a)). We also see that $\hat{J}_n$ for '$c = 8, r = 2$' is close to that for '$c = 4, r = 1$', as randomly mapping to 2 out of 8 sets is similar to mapping to 1 out of 4 cache sets. Our evaluation results suggests that SCATTERCACHE and PHANTOMCACHE eliminate side-channel leakage when there is no shared memory and largely restrict it when there is shared memory, if the address-to-cache mapping is random and remains unknown to the attacker.

5.4.4(b) Declassifying the memory-to-cache mapping When $\mathsf{I}(\mathsf{M})$ is unknown to the attacker, our previous analysis shows that cache-based side channels are mitigated. Werner et al. [93] also discuss the possibility of this mapping being disclosed to the attacker, however, through



(a) ScatterCache (b) PhantomCache

Figure 5.12: Memory sharing disabled (Prime+Probe attack), $\Delta(\text{'info'}) \leftarrow \mathsf{I}(\mathsf{M})$ (or $\mathsf{I}(\mathsf{M}^r)$)



(a) ScatterCache (b) PhantomCache

Figure 5.13: Memory sharing enabled (Flush+Reload attack), $\Delta(\text{'info'}) \leftarrow \mathsf{I}(\mathsf{M})$ (or $\mathsf{I}(\mathsf{M}^r)$)

```
1: function MODEXP(b,d)
2:     e ← 1
3:     for i ← n to 1 do
4:         e ← e × e mod M
5:         if d_i ≠ 0 then
6:             e ← e × b[d_i]
7:         end if
8:     end for
9: return e
10: end function
```

(a) Algorithm

```
proc(C,I,S)
  li sp, 0x80000400
  li  a0,1
  li a2, M
  li a3, S(d_i)
.oneIteration:
  mulw  a0,a0,a0
  remw  a0,a0,a2
  beqz  a3,.NextIteration
  sll  a5,a3,2
  add  a5,sp,a5
  lw  a5,0(a5)
  mulw  a0,a0,a5
  remw  a0,a0,a2
.NextIteration:
```

(b) Assembly for one iteration

Figure 5.14: Sliding window modular exponentiation with window size W. $d_n \dots d_1$ is the private key d where each d_i $(i = 1, \dots, n)$ is a W-bit value.

a profiling procedure. If we declassify $\mathsf{I}(\mathsf{M})$, the interference $\hat{J}_n^\delta$ will increase: Fig. 5.12(a) shows $\hat{J}_n^\delta$ due to PRIME+PROBE attacks in this case, and Fig. 5.13(a) shows the impact of this declassification on FLUSH+RELOAD attacks.

Similarly, using $\Delta(\text{'info'}) \leftarrow \mathsf{I}(\mathsf{M}^r_{dom})$, we evaluate PHANTOMCACHE's leakage when the random mapping is declassified; results are shown in Fig. 5.12(b) and Fig. 5.13(b). Comparing Fig. 5.12(b) and Fig. 5.12(a), PHANTOMCACHE's leakage (measured by $\hat{J}_n^\delta$) for unshared memory is higher than SCATTERCACHE's when $r = 1$. The strength of PHANTOMCACHE is revealed when r increases, since it allows memory blocks to map to more than one cache set. Specifically, the leakage for SCATTERCACHE's '$c = 4$' is much less than PHANTOMCACHE's '$c = 4,\ r = 1$' but is similar to PHANTOMCACHE's '$c = 4,\ r = 2$'. However, PHANTOMCACHE with $r = 2$ provides weaker protection for FLUSH+RELOAD than PHANTOMCACHE with $r = 1$ and SCATTERCACHE.

5.4.5 Leaking exponent in modular exponentiation

The evaluations in Sec. 5.4.3 and Sec. 5.4.4 focused on whether the adversary could detect the victim's access to a particular memory block, which is a well-known vector of information leakage. To further demonstrate the utility of our framework in measuring this type of leakage, here we consider a classic example whereby the secret is not a memory address, but rather is a cryptographic secret that, due to the algorithm in use, can influence the victim's cache footprint.

The particular example we evaluate here is modular exponentiation as used in algorithms such as RSA. A textbook implementation of modular exponentiation uses a sliding-window method that is known to leak information in caches [98, 10]. As shown in Fig. 5.14(a), the algorithm leverages

some small powers $b[k]$ of a base b (where $k < 2^W - 1$) to compute a larger power. Accesses to those precomputed powers is determined by the window-sized segment d_i of the private key d in each loop iteration i. First, this procedure will leak via the cache whether d_i is zero. Second, since the precomputed elements are addressed by d_i, an attacker may identify up to $\log_2 c$ bits about d_i if those precomputed powers map to different cache sets.

To evaluate the one-round leakage of Fig. 5.14(a), we used the RISC-V assembly shown in Fig. 5.14(b) in BOOM with a 2-way, 8-set cache ($c = 8$). The $\hat{J}_n$ measure shown in Fig. 5.15(a) indicates that the amount of leakage for one loop iteration i is limited, when $W \leq 4$ and so the precomputed b only uses up to $4 \times 2^4 = 64$ bytes (i.e., one cache line). When $4 < W < 8$, the side channel will leak more about d_i when W increases. Thus, choosing $W = 4$ is the best choice to protect the secret in our cache configuration.

To further diagnose the cause of leakage, we generated the interference rules for $W = 1$, $W = 4$, and $W = 8$. When $W = 1$, we obtain a single rule with precision and recall of 1.0, namely

$$\mathsf{C}(\text{`load'})[0] \geq 1 \wedge \mathsf{C}(\text{`load'})[8] \geq 1$$

This has no S or S' related conjuncts, indicating that the 1-bit secret d_i is fully leaked when an attacker PRIMEs one cache set. In contrast, when $W = 4$, the top rules (precision of 1.0, recall ≥ 0.5) include some S or S' related conjuncts, constraining the secret value to be zero, e.g.,

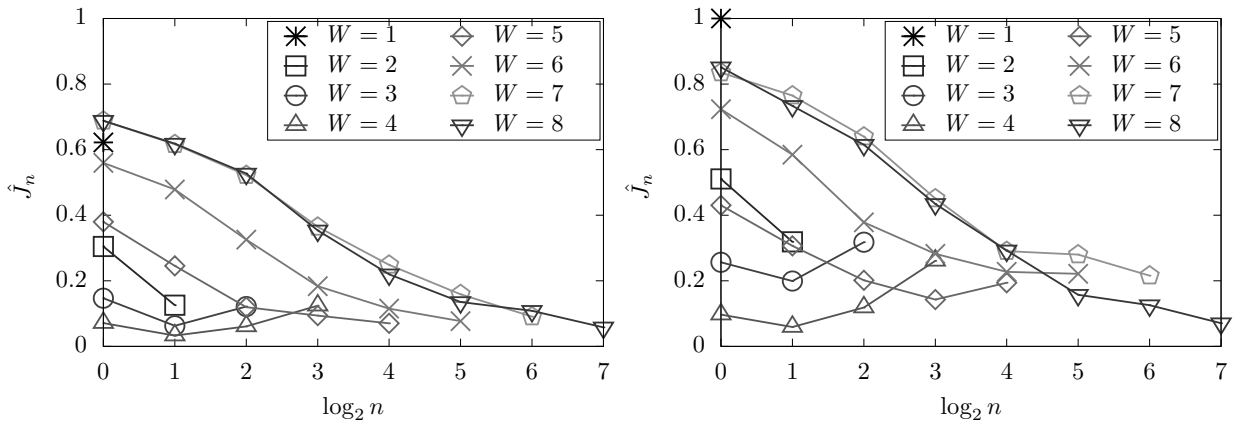
$$\mathsf{S}(d_i) < 1 \wedge \mathsf{C}(\text{`load'})[0] \geq 1 \wedge \mathsf{C}(\text{`load'})[8] \geq 1$$

That is, it only leaks whether it is zero or not for a 4-bit secret.

When $W > 4$, however, the most important cause of leakage changes from whether a memory access happens to which cache set is used by d_i. For example, when $W = 8$, one highly ranked rule (precision of 1.0, recall ≥ 0.04) is

$$\mathsf{S}'(d_i)[6] < 1 \wedge \quad \mathsf{S}'(d_i)[5] \geq 1 \wedge \quad \mathsf{S}'(d_i)[4] < 1$$
$$\wedge\ \mathsf{S}(d_i)[4] \geq 1 \wedge \mathsf{C}(\text{`load'})[10] \geq 1 \wedge \mathsf{C}(\text{`load'})[2] \geq 1$$

which indicates that the attacker can distinguish an $\mathsf{S}'(d_i)$ with $\mathsf{S}'(d_i)[4:6] = 2$ from an $\mathsf{S}(d_i)$ with



(a) Symbolic C('load') (b) $\forall \ell : \mathsf{C}$('load')$[\ell] = 1$

Figure 5.15: $\hat{J}_n$ for MODEXP in 2-way, 8-set cache

$\mathsf{S}(d_i)[4:6] \in \{1, 3, 5, 7\}$ if the attacker PRIMEs cache set 2. Similar to the analysis in Sec. 5.4.3(a) , rules for $W = 8$ illustrate that an attacker can reveal the cache set used by the victim (e.g., secret bits 4-6) when priming all cache sets.

5.4.6 Cache-based side channels in speculative execution

SPECTRE and its variants have received widespread attention in recent years. In a SPECTRE attack, a CPU predicts the outcome of a conditional branch and executes instructions based on that prediction to reduce delays incurred by those instructions if its prediction was correct. However, even if the prediction is incorrect, then some changes to the hardware state caused by speculative execution will persist even after the mispredicted computations have been discarded. These changes propagate information to exploitable side channels (cache-based side channels), allowing the attacker to steal it.

To explore such leaks using our framework, we used the software pseudocode in Fig. 5.16(b) and Fig. 5.16(c), each of which accesses an element of array arr2 at a secret index arr1[offset]. The bounds check on offset is dependent on reading arr1.size from memory in Fig. 5.16(b) and on a complex sequence of computations in Fig. 5.16(c). In the latter case, speculative execution might leak arr1[offset] through cache-based side channels, i.e., by bringing arr2[(arr1[offset] $\times$ 64) & 1023] into cache. Fig. 5.16(e) shows an important snippet of RISC-V assembly for Fig. 5.16(c) running above BOOM with a 2-way, 8-set cache. To evaluate the software snippet in Fig. 5.16(b), we change the block denoted by `.complexDependency` (Lines 10–16) with the `.shortDependency` in

```
conditionalAccess(offset, arr1.size)
  if (offset < arr1.size)
    tmp ← arr2[(arr1[offset] × 64) & 1023]
    declassify(arr1[offset])
```
(a) Conditional memory access

```
victimFunc(offset,secret)
  arr1[offset] ← secret
  read arr1.size from memory;
  conditionalAccess(offset, arr1.size)
```
(b) No bounds check bypass

```
victimFunc(offset,secret,arr1.size)
  arr1[offset] ← secret
  arr1.size← (arr1.size × 257) mod 256
  arr1.size← (arr1.size × 257) mod 256
  conditionalAccess(offset, arr1.size)
```
(c) Bounds check bypass

```
1 .shortDependency:
2   lbu  a0, 0x100(t3)
```
(d) Short speculation

```
 1 proc(C,I,S)
 2 .prepareData:
 3   li a0, I('arr1.size')
 4   li a1, C('offset')
 5   li a2, S('secret')
 6   //t3 ← arr1.addr
 7   //t4 ← arr2.addr
 8   add a3, t3, a1
 9   sb s2, 0(a3)
10 .complexDependency:
11   li  t1,0x101
12   li  t2,0x100
13   mul  a4,a0,t1
14   remuw  a4,a4,t2
15   mul  a4,a4,t1
16   remuw  a0,a4,t2
17 .conditionalAccess:
18   bleu  a0,a1,.end
19   add  t3,t3,a1
20   lbu  a3,0x0(t3)
21   sll  a3,a3,6
22   and  a3,a3,0x3ff
23   add  a3,t4,a3
24   lbu  a4,0(a3)
25 .end:
```
(e) Long speculation

Figure 5.16: Speculative execution example. Assembly in (e) is snippet from compilation of pseudocode in (c). Replacing lines 10–16 with (d) gives the analogous assembly for the pseudocode in (b).

Fig. 5.16(d). Furthermore, we evaluated a mitigation similar to `lfence` [1], by adding a RISC-V instruction '`fence r,r`' just after Line 18 in Fig. 5.16(e).

We assume the attacker can control the offset value C('offset'), train the *GShare* branch predictor C('bpd') shown in Fig. 5.5, and use FLUSH+RELOAD to observe O('hit'). The attacker can use the FLUSH+RELOAD-style attacks to precisely determine the index into arr2 if arr2 is shared and thus four bits of arr1[offset]. Note that the secret value S('secret') is assigned to arr1[offset] as the first step of Fig. 5.16(b) and Fig. 5.16(c). We presume that I('arr1.size') is an attacker-known but not controlled variable; thus, we include it as one output parameters as well, i.e., O('arr1.size') ← I('arr1.size').

As shown in Fig. 5.17, the $\hat{J}_n$ measures for 'ShortSpec' (denoting Fig. 5.16(d)) and 'Fence' are somewhat similar to that for 'LongSpec' (denoting Fig. 5.16(e))—contrary to what intuition would suggest. This counterintuitive result is due to the fact that leakage from *in-bounds* array accesses is also being counted. By declassifying in-bounds array elements (i.e., declassifying arr1[offset] if C('offset') < I('arr1.size')), we obtain a better picture of when leakage occurs. Specifically, when measuring the leakage with declassification of in-bounds array elements, $\hat{J}_n^{\delta}$ indicates that both *proc*

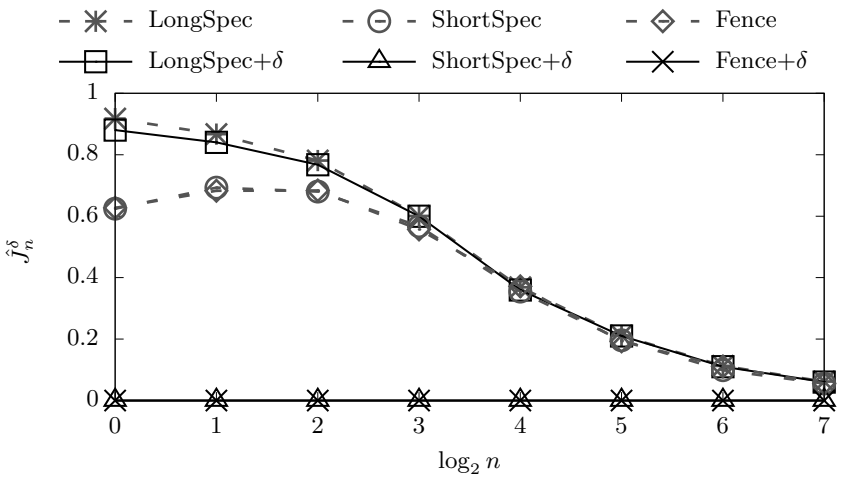


Figure 5.17: $\hat{J}_n^\delta$ for SPECTRE in different procedures

with the short dependency ('ShortSpec+δ') and *proc* with the `fence` mitigation ('Fence+δ') do not leak out-of-boundary memory contents, while the *proc* with the longer dependency ('LongSpec+δ') continues to leak secret data and indeed, is just slightly lower than 'complexDepend'.

In generating interference rules for *proc* with a long speculation (Fig. 5.16(e)), the linear feature

$$\begin{aligned} L_0 &= 0.005 \times \mathsf{S}(\text{`secret'}) - 0.003 \times \mathsf{S}'(\text{`secret'}) - \ 0.494 \times \mathsf{C}(\text{`offset'}) + 0.496 \times \mathsf{I}(\text{`arr1.size'}) \\ &\approx 0.5 \times \mathsf{I}(\text{`arr1.size'}) - 0.5 \times \mathsf{C}(\text{`offset'}) \end{aligned}$$

and specifically the conjunct $L_0 < 1$ appears in many of the top ranked rules. Using the approximation of L_0 above, $L_0 < 1$ implies that $\mathsf{I}(\text{`arr1.size'}) < \mathsf{C}(\text{`offset'}) + 2$, and so the offset is indeed out-of-bounds. An example top-ranked rule with precision 1.0 and recall 0.30 is

$$L_0 < 1 \wedge \mathsf{C}(\text{`bpd\{0\}.state'})[1] < 1 \wedge \ |\mathsf{S}(\text{`secret'})[2] - \mathsf{S}'(\text{`secret'})[2]| \geq 1$$

This rule indicates that an attacker can determine the third bit of the secret when the second bit of the state of the prediction entry $\mathsf{C}(\text{`bpd\{0\}.state'})$ is 0 ('strongly untaken') or 1 ('weakly untaken'). Analogous rules appear in the list for each of bits 0-2 and 4 of the secret. Other highly ranked rules

(also with precision 1.0 and recall 0.30) are

$$L_0 < 1 \wedge \mathsf{C}(\text{`bpd\{0\}.CFI'})[0] \geq 1 \wedge \ |\mathsf{S}(\text{`secret'})[0] - \mathsf{S}'(\text{`secret'})[0]| \geq 1 \tag{5.20}$$

$$L_0 < 1 \wedge \mathsf{C}(\text{`bpd\{0\}.CFI'})[1] < 1 \wedge \ |\mathsf{S}(\text{`secret'})[3] - \mathsf{S}'(\text{`secret'})[3]| \geq 1 \tag{5.21}$$

Rule (5.20) leaks the first bit of the secret when the 'CFI' value (i.e., $\mathsf{C}(\text{`bpd\{0\}.CFI'})$) in the prediction entry is 1 or 3, and (5.21) leaks the fourth bit when the 'CFI' value is 0 or 1. In these cases, the 'CFI' value does not match the CFI portion of the instruction address (i.e., the address of Line 18 in Fig. 5.16(e)), which was `0x800000800` + `0x44` (= `0b0 10 00100`), yielding a CFI portion of `0b 10` and *bidx* of `0b00`. Because of the mismatch on CFI value, $\mathsf{C}(\text{`bpd\{0\}.state'})$ is ignored and so speculation will not execute Lines 19–24. Though (5.20) and (5.21) are specific to the first or fourth bit of the secret, respectively, analogous rules appear for each of bits 0-3.

We have performed this evaluation using earlier BOOM versions and noticed that the out-of-bounds leakage was partially eliminated in version 2.2.3.[3] Since that version, the miss handling (MSHR) module of the L1 cache tracks branch prediction results and discards the pending cache refill request if a misprediction is detected before the refill commit. This change prevents the bounds check bypass in Fig. 5.16(b).

5.4.7 Performance

In DINoMe, we have four important components: an automated logical formula generator (Sec. 5.3.1), a model counter (Sec. 5.3.3), a sampler (Sec. 5.3.4), and a rule learner (Sec. 5.2.2). This section reports the time costs in the first three stages for all case studies we have evaluated. We performed those experiments on a DELL PowerEdge R815 server with 2.3GHz AMD Opteron 6376 processors and 128GB memory.

The time to generate the logical postcondition is primarily influenced by the number of RISC-V BOOM cycles represented by that postcondition, as we incrementally compose the formula cycle by cycle. Computing Π_{proc} required 20-40 minutes for the memory accessing experiments (100 cycles)

[3]In BOOM version 2.2.1, the victim program described in Fig. 5.16(b) also suffers the out-of-bounds leakage and thus has 'shortDepend' close to 'complexDepend' and 'shortDepend+declass' close to 'complexDepend+declass'.

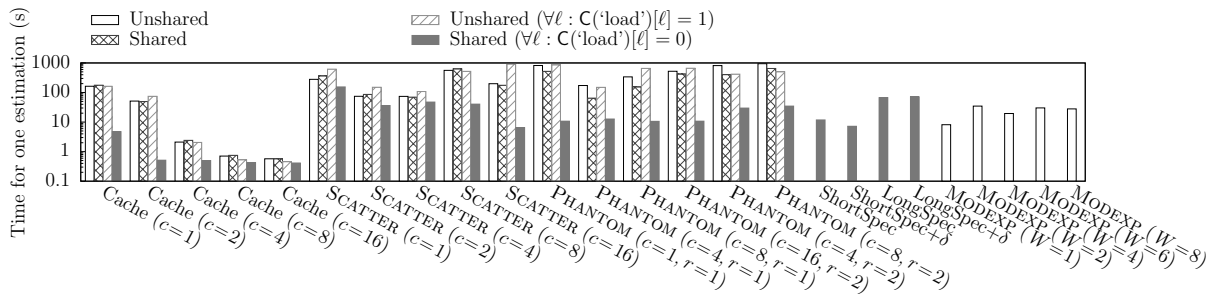


Figure 5.18: Time used in one estimation of $\hat{J}^{\delta}(S, S')$

in Sec. 5.4.3 and Sec. 5.4.4; 45 minutes for the modular exponentiation experiments (120 cycles) in Sec. 5.4.5; and around 2 hours for the SPECTRE experiments (150 cycles) in Sec. 5.4.6.

Fig. 5.18 shows the runtime to compute *one* estimate of $\hat{J}(S, S')$ or $\hat{J}^{\delta}(S, S')$ in the model counting process; note the logarithmic y-axis. Specifically, counting for cache-based side channels in SCATTERCACHE and PHANTOMCACHE are much more expensive than others, where one estimate requires up to 16 minutes. The difficulty in counting for SCATTERCACHE (denoted by 'SCATTER') and PHANTOMCACHE (denoted by 'PHANTOM') is due to the large size of their counting variables. For SCATTERCACHE, the memory-to-cache mapping uses $log_2(c) \times w$ bits per domain per memory block for 32 memory blocks. Specifically, the 8-way 2-set SCATTERCACHE (denoted by 'SCATTER $(c = 2)$'), uses 512 bits to represent $\mathsf{I}(\mathsf{M})$, which means the counting process would add hundreds of XOR constraints to compute one estimate, which greatly increases the difficulty to find a feasible solution. To obtain the sample sets $\hat{IS}$ and $\hat{NS}$, the sampling process generates a tuple in $\hat{IS}$ or $\hat{NS}$ within seconds, as illustrated in Fig. 5.19.

Our reported results reflect estimations of $\hat{J}(S, S')$ or $\hat{J}^{\delta}(S, S')$ for at least 100 S, S' pairs per n, and we sampled up to 100,000 tuples in $\hat{IS}$ and $\hat{NS}$. These estimations and samplings are trivially parallelizable and so, with horizontal scaling, can be performed in total times approaching those in Fig. 5.18 and Fig. 5.19 to the extent budget allows.

5.5 Limitations

Despite the scalability improvements represented by DINOME specifically for analyzing processor designs, it still has limitations. First, due to the complexity of hardware logic, generating the postcondition $\Pi_{proc}(\mathsf{C}, \mathsf{I}, \mathsf{S}, \mathsf{O})$ for a *proc* representing both the OS and the application would require more CPU cycles than the number to which we have been able to scale DINOME thus far.

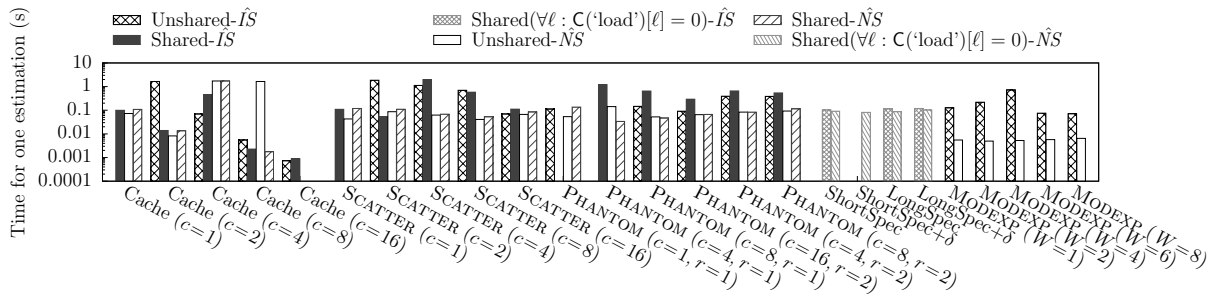


Figure 5.19: Time used in generating one tuple in $\hat{NS}$ or $\hat{IS}$

The DINoMe workloads described in this chapter represent a tradeoff, using a sequence of opcodes with concretized operations and selected symbolic operands above a partially symbolic hardware specification. To evaluate with more complicated software, a possible solution is to highly concretize the initial hardware state (especially for the memory and cache states) or highly concretize the software, at the cost of possibly missing some potential leakage that remains hidden due to this concretization.

A second limitation of DINoMe, and specifically of its generation of interpretation rules to explain leakage, is that the interpretation rules may not be complete, for two reasons. First, the interpretation rules might skip a rule that only covers a small portion of leakage samples (i.e., with low recall). A possible solution to address this source of incompleteness is to declassify the sources of leakage exposed in the inference rules that *are* learned, and then rerun the learning process again. Second, the conditions that result in leakage might be more complicated than can be learned using decision trees built using local linear classifiers. To address this incompleteness, alternative learning methods might be tried, though doing so while retaining interpretability will be a challenge.

5.6 Summary

Scaling high-fidelity, static noninterference measurement to complex computations has been a challenge since the introduction of noninterference in the 1980's [44]. We believe that we have advanced the state-of-the-art in this area both generally and specifically for its application to processor designs. Certain innovations in our DINoMe framework, such as the cycle-by-cycle construction of the logical postcondition for processor execution, are specific to processor designs. Others, such as our methods for declassification and interpreting leakage results, are not. Together, however, they permit the measurement of leakage in complex scenarios, as we demonstrated through

using DINoMe to analyze leakage due to speculative execution in the BOOM core and of published defenses to mitigate it. Our analysis enables comparisons between defenses to discover, e.g., the processor and defense parameterizations where one defense outperforms the other. Though the performance of DINoMe suggests that static measurement of noninterference for processors is still too time-intensive for highly interactive use, it is fast enough to permit multiple analysis iterations per day in many cases. And through its improvements in declassification and interpretability, it substantially facilitates human understanding of its measurement results.

CHAPTER 6: CONCLUSION

Any computation with insecure information flows is potentially vulnerable to side-channel attacks. Noninterference was conceived as a requirement to eliminate any such flows. In practice, however, absolute noninterference can rarely be achieved. This dissertation has thus made contributions toward the development of *measuring* noninterference for real-world programs.

First, we explored noninterference assessment using empirical evaluations against cache-based side-channel attacks. Relying on these empirical evaluations, we demonstrated CacheBar's effectiveness in defending against cache-based side channels in `LLC`s. However, model checking revealed that empirical assessment was not enough, as it failed to capture interference not triggered by the concrete experiments. This lesson motivated the use of formal approaches for assessing noninterference more holistically.

Second, we suggested a static method for measuring interference from actual codebases. One contribution of our measurement is its formulation of interference as the distinguishability of two sets of secret values. This novel metric supports noninterference measurement in multiple dimensions, reflecting how often secrets leak when the size n of secret sets is small and how much is leaked when n is large. Case studies showed that our measurement framework has moderate runtime costs, which range from minutes to days depending on the workload and the computation resources (i.e., parallelization is possible).

Third, we extended the static framework to relatively complicated computations including the processor on which they execute, and implemented them in DINoMe. By leveraging the declassification and interpretation capabilities of DINoMe, we measured and explained hardware-software vulnerabilities. DINoMe analyzes a sequence of instructions running for hundreds of cycles above the BOOM processor. Logical rules sorted by their precision and recall values explain the sources of leakage without forcing the analyst to diagnose the leakage from the measurement value alone.

We demonstrated the possibility of using our frameworks to measure and interpret unintended leakage in practice using our case studies on side-channel leakage through shared caches, traffic

analysis, adaptive compression algorithms, shared TCP network counters, sliding-window modular exponentiation algorithms, and speculative executions. We hope that these demonstrations of noninterference measurement will bring quantitative information flow (QIF) closer to practice and help analysts develop better mitigations.

REFERENCES

[1] Intel analysis of speculative execution side channels. Technical report, Intel Corp., Jan 2018.

[2] P. A, Y. Rodeh, O. Strichman, and M. Siegel. The small model property: How small can it be? *Information and Computation*, 178(1):279–293, 2002.

[3] O. Aciiçmez. Yet another microarchitectural attack: Exploiting I-cache. In *ACM Workshop on Computer Security Architecture*, pages 11–18, 2007.

[4] J. Alawatugoda, D. Stebila, and C. Boyd. Protecting encrypted cookies from compression side-channel attacks. In *Financial Cryptography and Data Security*, pages 86–106, 2015.

[5] A. Arcangeli, I. Eidus, and C. Wright. Increasing memory density by using KSM. In *Linux Symposium*, pages 19–28, 2009.

[6] R. A. Aziz, G. Chu, C. Muise, and P. Stuckey. #∃SAT: Projected model counting. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 121–137. Springer, 2015.

[7] M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *30th IEEE Symposium on Security and Privacy*, pages 141–153, 2009.

[8] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In *International Conference on Integrated Formal Methods*, pages 1–20. Springer, 2004.

[9] A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *29th IEEE Symposium on Security and Privacy*, pages 339–353, 2008.

[10] D. J. Bernstein, J. Breitner, D. Genkin, L. G. Bruinderink, N. Heninger, T. Lange, C. V. Vredendaal, and T. Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 555–576. Springer, 2017.

[11] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *37th IEEE Symposium on Security and Privacy*, pages 987–1004, 2016.

[12] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224, 2008.

[13] Y. Cao, Z. Qian, Z. Wang, T. Dao, S. V. Krishnamurthy, and L. M. Marvel. Off-path TCP exploits: Global rate limit considered dangerous. In *25th USENIX Security Symposium*, pages 209–225, 2016.

[14] C. Celio, P. Chiu, B. Nikolic, P. D. A, and K. Asanovic. BOOMv2: an open-source out-of-order RISC-V core. In *1st Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017.

[15] S. Chakraborty, K. S. Meel, and M. Y. Vardi. A scalable approximate model counter. In *Principles and Practice of Constraint Programming*, volume 8124 of *LNCS*, pages 200–216, 2013.

[16] P. Chapman and D. Evans. Automated black-box detection of side-channel vulnerabilities in web applications. In *18th ACM Conference on Computer and Communications Security*, pages 263–274, 2011.

[17] Q. A. Chen, Z. Qian, Y. J. Jia, Y. Shao, and Z. M. Mao. Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks. In *22nd ACM Conference on Computer and Communications Security*, pages 388–400, 2015.

[18] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *31st IEEE Symposium on Security and Privacy*, pages 191–206, 2010.

[19] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *22rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, 2016.

[20] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–278, 2011.

[21] S. Chong and A. C. Myers. Security policies for downgrading. In *11th ACM conference on Computer and communications security*, pages 198–209, 2004.

[22] D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. *Electronic Notes in Theoretical Computer Science*, 59(3):238–251, 2002.

[23] D. Clark, S. Hunt, and P. Malacaria. Quantitative information flow, relations and polymorphic types. *Journal of Logic and Computation*, 15(2):181–199, 2005.

[24] D. Clark, S. Hunt, and P. Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15(3):321–371, 2007.

[25] M. R. Clarkson, A. C. Myers, and F. B. Schneider. Belief in information flow. In *18th IEEE Computer Security Foundations Workshop*, pages 31–45, 2005.

[26] W. W. Cohen and Y. Singer. A simple, fast, and effective rule learner. *16th AAAI Conference on Artificial Intelligence*, 99:335–342, 1999.

[27] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *30th IEEE Symposium on Security and Privacy*, pages 45–60, 2009.

[28] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *22nd Network and Distributed System Security Symposium*, pages 8–11, 2015.

[29] D. E. R. Denning. *Cryptography and data security*, volume 112. Addison-Wesley Reading, 1982.

[30] J. Dike. User-mode Linux. In *5th Annual Linux Showcase & Conference*, pages 3–14, 2001.

[31] Y. Dodis, R. Ostrovsky, L. Reyzin, and A. Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. *SIAM Journal on Computing*, 38(1):97–139, 2008.

[32] B. Dutertre. Solving exists/forall problems with yices. In *Workshop on Satisfiability Modulo Theories*, 2015.

[33] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography*, pages 265–284, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[34] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail. In *33rd IEEE Symposium on Security and Privacy*, pages 332–346, 2012.

[35] L. Elizaveta and P. Bickel. The earth mover's distance is the Mallows distance. In *8th International Conference on Computer Vision*, pages 251–256, 2001.

[36] J. Fan. Local linear regression smoothers and their minimax efficiencies. *The Annals of Statistics*, pages 196–216, 1993.

[37] R. Fan, K. Chang, C. Hsieh, X. Wang, and C. Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9(Aug):1871–1874, 2008.

[38] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. Suh. Verification of a practical hardware security architecture through static information flow analysis. In *22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 555–568, 2017.

[39] M. Fokkema. Fitting prediction rule ensembles with R package pre. *Journal of Statistical Software*, 92(12):1–30, 2020.

[40] J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, pages 1189–1232, 2001.

[41] J. H. Friedman and B. E. Popescu. Predictive learning via rule ensembles. *The Annals of Applied Statistics*, 2(3):916–954, 2008.

[42] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. *ACM SIGPLAN Notices*, 39(1):186–197, 2004.

[43] R. Giacobazzi and I. Mastroeni. Abstract non-interference: a unifying framework for weakening information-flow. *ACM Transactions on Privacy and Security (TOPS)*, 21(2):1–31, 2018.

[44] J. A. Goguen and J. Meseguer. Security policies and security models. In *3rd IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[45] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting. *Handbook of Satisfiability*, pages 633–654, 2008.

[46] J. W. Gray. Toward a mathematical foundation for information flow security. In *12nd IEEE Symposium on Security and Privacy*, pages 21–34, 1991.

[47] D. Gruss, C. Maurice, K. Wagner, S. Mangard, U. Zurutuza, and R. J. Rodríguez. Flush+Flush: A stealthier last-level cache attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume abs/1511.04594, pages 279–299. Springer, 2016.

[48] X. Guo, R. G. Dutta, J. He, M. M. Tehranipoor, and Y. Jin. Qif-verilog: Quantitative information-flow based hardware description languages for pre-silicon security assessment. In *IEEE International Symposium on Hardware Oriented Security and Trust*, pages 91–100, 2019.

[49] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2010.

[50] G. Irazoqui, T. Eisenbarth, and B. Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing—and its application to AES. In *36th IEEE Symposium on Security and Privacy*, pages 591–604, 2015.

[51] A. Ivrii, S. Malik, K. S. Meel, and M. Y. Vardi. On computing minimal independent support and its applications to sampling and counting. *Constraints*, 21(1):41–58, 2016.

[52] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill. Generic attacks on secure outsourced databases. In *23rd ACM Conference on Computer and Communications Security*, pages 1329–1340, 2016.

[53] J. Kelsey. Compression and information leakage of plaintext. In *9th International Workshop on Fast Software Encryption*, pages 263–276, 2002.

[54] G. Keramidas, A. Antonopoulos, D. N. Serpanos, and S. Kaxiras. Non deterministic caches: A simple and effective defense against side channel attacks. *Design Automation for Embedded Systems*, 12(3):221–230, 2008.

[55] T. Kim, M. Peinado, and G. Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *21st USENIX Security Symposium*, pages 189–204, 2012.

[56] V. Klebanov, N. Manthey, and C. Muise. SAT-based analysis and quantification of information flow in programs. In *International Conference on Quantitative Evaluation of Systems*, pages 177–192. Springer, 2013.

[57] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, and T. Prescher. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy*, pages 1–19, 2019.

[58] R. Könighofer. A fast and cache-timing resistant implementation of the AES. In *Topics in Cryptology – CT-RSA*, pages 187–202, 2008.

[59] B. Köpf and D. Basin. An information-theoretic model for adaptive side-channel attacks. In *14th ACM Conference on Computer and Communications Security*, pages 286–296, 2007.

[60] B. Köpf and A. Rybalchenko. Approximation and randomization for quantitative information-flow analysis. In *23rd IEEE Computer Security Foundations Symposium*, pages 3–14, 2010.

[61] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *33rd ACM Conference on Programming Language Design and Implementation*, pages 193–204, 2012.

[62] J. Lagniez, E. Lonca, and P. Marquis. Improving model counting by leveraging definability. In *25th International Joint Conference on Artificial Intelligence*, pages 751–757, 2016.

[63] J. Lagniez and P. Marquis. On preprocessing techniques and their impact on propositional model counting. *Journal of Automated Reasoning*, 58(4):413–481, 2017.

[64] Linux blind TCP spoofing vulnerability. `http://www.securityfocus.com/bid/580/info`, 1999.

[65] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, S. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium*, pages 973–990, 2018.

[66] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *22nd IEEE Symposium on High Performance Computer Architecture*, pages 406–418, 2016.

[67] F. Liu and R. B. Lee. Random fill cache architecture. In *47th IEEE/ACM International Symposium on Microarchitecture*, pages 203–215, 2014.

[68] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *36th IEEE Symposium on Security and Privacy*, pages 605–622, 2015.

[69] P. Malacaria. Assessing security threats of looping constructs. In *34th ACM Symposium on Principles of Programming Languages*, pages 225–235, 2007.

[70] N. Manthey. Coprocessor 2.0–a flexible CNF simplifier. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 436–441. Springer, 2012.

[71] P. Mardziel, M. S. Alvim, M. Hicks, and M. R. Clarkson. Quantifying information flow for dynamic secrets. In *35th IEEE Symposium on Security and Privacy*, pages 540–555, 2014.

[72] R. MORRIS. A weakness in the 4.2 BSD Unix TCP/IP software. *AT&T Bell Labs, Tech. Rep. Comput. Sci.*, 117, 1985.

[73] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *Topics in Cryptology – CT-RSA*, pages 1–20. Springer, 2006.

[74] R. Owens and W. Wang. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In *30th IEEE International Performance Computing and Communications Conference*, pages 1–8, 2011.

[75] C. Percival. Cache missing for fun and profit. In *BSDCan*, 2005.

[76] Q. Phan, L. Bang, C. S. Păsăreanu, P. Malacaria, and T. Bultan. Synthesis of adaptive side-channel attacks. In *30th IEEE Computer Security Foundations Symposium*, pages 328–342, 2017.

[77] Q. Phan and P. Malacaria. Abstract model counting: A novel approach for quantification of information leaks. In *9th ACM Symposium on Information, Computer and Communications Security*, pages 283–292, 2014.

[78] C. S. Păsăreanu, Q. Phan, and P. Malacaria. Multi-run side-channel analysis using symbolic execution and max-SMT. In *29th IEEE Computer Security Foundations Symposium*, pages 387–400, 2016.

[79] Z. Qian, Z. M. Mao, and T. Xie. Collaborative TCP sequence number inference attack – how to crack sequence number under a second. In *19th ACM Conference on Computer and Communications Security*, pages 593–604, 2012.

[80] S. Raikin, J. D. Sager, Z. Sperber, E. Krimer, O. Lempel, S. Shwartsman, A. Yoaz, and O. Golz. Tracking mechanism coupled to retirement in reorder buffer for indicating sharing logical registers of physical register in record indexed by logical register, 2014. US Patent 8,914,617.

[81] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium*, pages 431–446, 2015.

[82] M. T. Ribeiro, S. Singh, and C. Guestrin. Anchors: High-precision model-agnostic explanations. In *32rd AAAI Conference on Artificial Intelligence*, pages 1527–1535, 2018.

[83] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *International Symposium on Software Security*, pages 174–191. Springer, 2003.

[84] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, pages 517–548, 2009.

[85] S. Sanfilippo. Small strings compression library. `https://github.com/antirez/smaz`, 2009.

[86] G. Smith. On the foundations of quantitative information flow. In *International Conference on Foundations of Software Science and Computational Structures*, pages 288–302. Springer, 2009.

[87] G. Smith. Quantifying information flow using min-entropy. In *8th International Conference on Quantitative Evaluation of Systems*, pages 159–167, 2011.

[88] M. Soos and K. S. Meel. BIRD: Engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In *36th AAAI Conference on Artificial Intelligence*, volume 33, pages 1592–1599, 2019.

[89] M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, pages 244–257, 2009.

[90] Q. Tan, Z. Zeng, K. Bu, and K. Ren. Phantomcache: Obfuscating cache conflicts with localized randomization. In *27th Network and Distributed System Security Symposium*, 2020.

[91] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *34th International Symposium on Computer Architecture*, pages 494–505, 2007.

[92] Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *41st IEEE/ACM International Symposium on Microarchitecture*, pages 83–93, 2008.

[93] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard. ScatterCache: Thwarting cache attacks via cache set randomization. In *28th USENIX Security Symposium*, pages 675–692, Santa Clara, CA, 2019.

[94] C. Wolf. Yosys open synthesis suite. `http://www.clifford.at/yosys/`.

[95] Y. Yarom and K. E. Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium*, pages 719–732, 2014.

[96] K. Zhang, Z. Li, R. Wang, X. Wang, and S. Chen. Sidebuster: Automated detection and quantification of side-channel leaks in web application development. In *17th ACM Conference on Computer and Communications Security*, pages 595–606, 2010.

[97] R. Zhang, C. Deutschbein, P. Huang, and C. Sturton. End-to-end automated exploit generation for validating the security of processor designs. In *51st IEEE/ACM International Symposium on Microarchitecture*, pages 815–827, 2018.

[98] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *19th ACM Conference on Computer and Communications Security*, pages 305–316, 2012.

[99] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *21st ACM Conference on Computer and Communications Security*, pages 990–1003, 2014.

[100] Z. Zhou, Z. Qian, M. K. Reiter, and Y. Zhang. Static evaluation of noninterference using approximate model counting. In *39th IEEE Symposium on Security and Privacy*, pages 514–528, 2018.

[101] Z. Zhou, M. K. Reiter, and Y. Zhang. A software approach to defeating side channels in last-level caches. In *23rd ACM Conference on Computer and Communications Security*, pages 871–882, 2016.